HIGH SPEED AND LARGE SCALE
SCIENTIFIC COMPUTING
Advances in Parallel Computing

This book series publishes research and development results on all aspects of parallel computing. Topics may include one or more of the following: high speed computing architectures (Grids, clusters, Service Oriented Architectures, etc.), network technology, performance measurement, system software, middleware, algorithm design, development tools, software engineering, services and applications.

Series Editor:
Professor Dr. Gerhard R. Joubert

Volume 18
Recently published in this series
Vol. 17. F. Xhafa (Ed.), Parallel Programming, Models and Applications in Grid and P2P Systems
Vol. 16. L. Grandinetti (Ed.), High Performance Computing and Grids in Action

Volumes 1–14 published by Elsevier Science.

ISSN 0927-5452
High Speed and Large Scale Scientific Computing

Edited by

Wolfgang Gentzsch
DEISA Project and Open Grid Forum, Germany

Lucio Grandinetti
Center of Excellence on High Performance Computing, University of Calabria, Italy

and

Gerhard Joubert
Institute of Informatics, Clausthal University of Technology, Germany

IOS Press
Amsterdam • Berlin • Tokyo • Washington, DC
Preface

During the last decade parallel computing technologies transformed main stream computing. The majority of notebooks and standard PC’s today incorporate multi processor chips with up to four processors. This number is expected to soon reach eight and more. These standard components allow the construction of high speed parallel systems in the petascale range at a reasonable cost. The number of processors incorporated in such systems is of the order of $10^4$ to $10^6$.

Due to the flexibility offered by parallel systems constructed with commodity components, these can easily be linked through wide area networks, for example the Internet, to realise Grids or Clouds. Such networks can be accessed by a wide community of users from many different disciplines to solve compute intensive and/or data intensive problems requiring high speed computing resources.

The problems associated with the efficient and effective use of such large systems were the theme of the biannual High Performance Computing workshop held in July 2008 in Cetraro, Italy. A selection of papers presented at the workshop are combined with a number of invited contributions in this book.

The papers included cover a range of topics, from algorithms and architectures to Grid and Cloud technologies to applications and infrastructures for e-science.

The editors wish to thank all the authors for preparing their contributions as well as the reviewers who supported this effort with their constructive recommendations.

Wolfgang Gentzsch  Gerhard Joubert  Lucio Grandinetti
Germany  Netherlands/Germany  Italy

Date: 2009-09-10
Reviewers

Hans-Joachim Bungartz  Germany
Ewa Deelman  USA
Efstratios Gallopoulos  Greece
Adriana Iamnitchi  USA
Chris Jesshope  Netherlands
Odej Kao  Germany
Nicolas Kourtellis  USA
Janusz Kowalik  USA
Dieter Kranzlmüller  Germany
Marcel Kunze  Germany
Thomas Lippert  Germany
Ignacio Martin Llorente  Spain
Andy Marsh  UK
Ralph Niederberger  Germany
Wilfried Philips  Belgium
Morris Riedel  Germany
Domenico Talia  Italy
Denis Trystram  France
Anette Weisbecker  Germany
Ramin Yahyapour  Germany
Contents

Preface v
Wolfgang Gentzsch, Gerhard Joubert and Lucio Grandinetti

Reviewers vi

Chapter 1. Algorithms and Scheduling

Scheduling for Numerical Linear Algebra Library at Scale 3
Jakub Kurzak, Hatem Ltaief, Jack J. Dongarra and Rosa M. Badia

Algorithms and Scheduling Techniques for Clusters and Grids 27
Anne Benoit, Loris Marchal, Yves Robert and Frédéric Vivien

Chapter 2. Architectures

High Performance Computing with FPGAs 55
Erik H. D’Hollander and Kristof Beyls

Nondeterministic Coordination Using S-Net 74
Alex Shafarenko

HPC Interconnection Networks: The Key to Exascale Computing 95
Jeffrey S. Vetter, Vinod Tipparaju, Weikuan Yu and Philip C. Roth

Chapter 3. GRID Technologies

Using Peer-to-Peer Dynamic Querying in Grid Information Services 109
Domenico Talia and Paolo Trunfio

Emulation Platform for High Accuracy Failure Injection in Grids 127
Thomas Herault, Mathieu Jan, Thomas Largillier, Sylvain Peyronnet, Benjamin Quetier and Franck Cappello

DEISA, the Distributed European Infrastructure for Supercomputing Applications 141
Wolfgang Gentzsch

UNICORE 6 – A European Grid Technology 157
Achim Streit, Sandra Bergmann, Rebecca Breu, Jason Daivandy, Bastian Demuth, André Giesler, Björn Hagemeier, Sonja Holl, Valentina Huber, Daniel Mallmann, Ahmed Shiraz Memon, Mohammad Shahbaz Memon, Roger Menday, Michael Rambadt, Morris Riedel, Mathilde Romberg, Bernd Schuller and Thomas Lippert
Chapter 4. Cloud Technologies

Cloud Computing for on-Demand Grid Resource Provisioning 177
Ignacio M. Llorente, Rafael Moreno-Vozmediano and Rubén S. Montero

Clouds: An Opportunity for Scientific Applications? 192
Ewa Deelman, Bruce Berriman, Gideon Juve, Yang-Suk Kee, Miron Livny and Gurmeet Singh

Cloud Computing: A Viable Option for Enterprise HPC? 216
Mathias Dalheimer and Franz-Josef Pfreundt

Evidence for a Cost Effective Cloud Computing Implementation Based Upon the NC State Virtual Computing Laboratory Model 236
Patrick Dreher, Mladen A. Vouk, Eric Sills and Sam Averitt

Facing Services in Computational Clouds 251
Thijs Metsch, Luis M. Vaquero, Luis Rodero-Merino, Maik Lindner and Philippe Massonet

Aneka: A Software Platform for .NET Based Cloud Computing 267
Christian Vecchiola, Xingchen Chu and Rajkumar Buyya

Chapter 5. Information Processing and Applications

Building Collaborative Applications for System-Level Science 299
Marian Bubak, Tomasz Gubala, Marek Kasztelnik and Maciej Malawski

Parallel Data Mining from Multicore to Cloudy Grids 311
Geoffrey Fox, Seung-Hee Bae, Jaliya Ekanayake, Xiaohong Qiu and Huapeng Yuan

Processing of Large-Scale Biomedical Images on a Cluster of Multicore CPUs and GPUs 341
Umit V. Catalyurek, Timothy D.R. Hartley, Olcay Sertel, Manuel Ujaldon, Antonio Ruiz, Joel Saltz and Metin Gurcan

System Level Accelerator with Blue Gene: A Heterogeneous Computing Model for Grand Challenge Problems 365
Tim David

Grid Computing for Financial Applications 380
Patrizia Beraldi, Lucio Grandinetti, Antonio Violi and Italo Epicoco

Chapter 6. HPC and GRID Infrastructures for e-Science

An Active Data Model 399
Tim Ho and David Abramson

The Evolution of Research and Education Networks and Their Essential Role in Modern Science 422
William Johnston, Evangelos Chaniotakis, Eli Dart, Chin Guok, Joe Metzger and Brian Tierney
The European Grid Initiative and the HPC Ecosystem

Per Öster

Grid and e-Science in Korea

Kihyeon Cho

Subject Index

Author Index
This page intentionally left blank
Chapter 1

Algorithms and Scheduling
This page intentionally left blank
Scheduling for Numerical Linear Algebra Library at Scale

Jakub KURZAK a,1, Hatem LTAIEF a, Jack J. DONGARRA a,b,c, Rosa M. BADIA d

a Department of Electrical Engineering and Computer Science, University of Tennessee
b Computer Science and Mathematics Division, Oak Ridge National Laboratory
c School of Mathematics & School of Computer Science, University of Manchester
d Barcelona Supercomputing Center - Centro Nacional de Supercomputación

Abstract. State-of-the-art dense linear algebra software, such as the LAPACK and ScaLAPACK libraries, suffer performance losses on multicore processors due to their inability to fully exploit thread-level parallelism. At the same time the coarse-grain dataflow model gains popularity as a paradigm for programming multicore architectures. This work looks at implementing classic dense linear algebra workloads, Cholesky factorization and QR factorization, using dynamic data-driven execution. Two emerging approaches to implementing coarse-grain dataflow are examined, the model of nested parallelism, represented by the Cilk framework, and the model of parallelism expressed through an arbitrary Direct Acyclic Graph, represented by the SMP Superscalar framework. Performance and coding effort are analyzed and compared against code manually parallelized at the thread level.

Keywords. task graph, scheduling, multicore, linear algebra, matrix factorization, Cholesky, QR

Introduction

The current trend in the semiconductor industry to double the number of execution units on a single die is commonly referred to as the multicore discontinuity. This term reflects the fact that existing software model is inadequate for the new architectures and existing code base will be incapable of delivering increased performance, possibly not even capable of sustaining current performance.

This problem has already been observed with state-of-the-art dense linear algebra libraries, LAPACK [1] and ScaLAPACK [2], which deliver a small fraction of peak performance on current multicore processors and multi-socket systems of multicore processors, mostly following Symmetric Multi-Processor (SMP) architecture.

The problem is twofold. Achieving good performance on emerging chip designs is a serious problem, calling for new algorithms and data structures. Reimplementing existing code base using a new programming paradigm is another major challenge, specifically in the area of high performance scientific computing, where the level of required skills makes the programmers a scarce resource and millions of lines of code are in question.

1Corresponding Author: Jakub Kurzak, 1122 Volunteer Blvd, Ste 413 Claxton, Knoxville, TN 37996-3450, United States; E-mail: kurzak@eecs.utk.edu
1. Background

In large scale scientific computing, targeting distributed memory systems, the recent push towards the PetaFlop barrier caused a renewed interest in Partitioned Global Address Space (PGAS) languages, such as Co-Array Fortran (CAF) [3], Unified Parallel C (UPC) [4] or Titanium [5], as well as emergence of new languages, such as Chapel (Cray) [6], Fortress (Sun) [7] and X-10 (IBM) [8], sponsored through the DARPA’s High Productivity Computing Systems (HPCS) program.

In more mainstream, server and desktop computing, targeting mainly shared memory system, the well known dataflow model is rapidly gaining popularity, where the computation is viewed as a Direct Acyclic Graph (DAG), with nodes representing computational tasks and edges representing data dependencies among them. The coarse-grain dataflow model is the main principle behind emerging multicore programming environments such as Cilk/Cilk++ [9], Intel® Threading Building Blocks (TBB) [10, 11], Tasking in OpenMP 3.0 [12, 13, 14, 15] and SMP Superscalar (SMPSs) [16].

All these frameworks rely on a very small set of extensions to common imperative programming languages such as C/C++ and Fortran and involve a relatively simple compilation stage and potentially much more complex runtime system.

The following sections provide a brief overview of these frameworks, as well as an overview of a rudimentary scheduler implemented using POSIX threads, which will serve as a baseline for performance comparisons.

Since tasking facilities available in Threading Building Blocks and OpenMP 3.0 closely resemble the ones provided by Cilk, Cilk is chosen as a representative framework for all three (also due to the reason that, same as SMPSs, it is available in open-source).

1.1. Cilk

Cilk was developed at the MIT Laboratory for Computer Science starting in 1994 [9]. Cilk is an extension of the C language with a handful of keywords (cilk, spawn, sync, inlet, abort) aimed at providing general-purpose programming language designed for multithreaded parallel programming. When the Cilk keywords are removed from Cilk source code, the result is a valid C program, called the serial elision (or C elision) of the full Cilk program. The Cilk environment employs a source-to-source compiler, which compiles Cilk code to C code, a standard C compiler, and a runtime system linked with the object code to provide an executable.

The main principle of Cilk is that the programmer is responsible for exposing parallelism by identifying functions free of side effects (e.g., access to global variables causing race conditions), which can be treated as independent tasks and executed in parallel. Such functions are annotated with the cilk keyword and invoked with the spawn keyword. The sync keyword is used to indicate that execution of the current procedure cannot proceed until all previously spawned procedures have completed and returned their results to the parent.

Distribution of work to multiple processors is handled by the runtime system. Cilk scheduler uses the policy called work-stealing to schedule execution of tasks to multiple processors. At run time, each processor fetches tasks from the top of its own stack - in First In First Out (FIFO) order. However, when a processor runs out of tasks, it picks another processor at random and "steals" tasks from the bottom of its stack - in Last In
First Out (LIFO) order. This way the task graph is consumed in a depth-first order, until a processor runs out of tasks, in which case it steals tasks from other processors in a breadth-first order.

Cilk also provides the mechanism of locks. The use of lock can, however, easily lead to deadlock. "Even if the user can guarantee that his program is deadlock-free, Cilk may still deadlock on the user’s code because of some additional scheduling constraints imposed by Cilk’s scheduler" [17]. In particular locks cannot be used to enforce parent-child dependencies between tasks.

Cilk is very well suited for expressing algorithms which easily render themselves to recursive formulation, e.g., divide-and-conquer algorithms. Since stack is the main structure for controlling parallelism, the model allows for straightforward implementations on shared memory multiprocessor systems (e.g., multicore/SMP systems). The simplicity of the model provides for execution of parallel code with virtually no overhead from scheduling.

1.2. OpenMP

OpenMP was born in the ’90s to bring a standard to the different directive languages defined by several vendors. The different characteristics of this approach: simplicity of the interface, use of a shared memory model, and the use of loosely-coupled directives to express the parallelism of a program, make it very well-accepted today. Due to new needs of the parallel applications, OpenMP has been recently extended to add, in its version 3.0, a tasking model that addresses new programming model aspects.

The new OpenMP directives allows the programmer to identify units of independent work (tasks), leaving the decision to how and when to execute them to the runtime system.

This gives the programmers a way of expressing patterns of concurrency that do not match the worksharing constructs defined in the OpenMP 2.5 specification. The main difference between Cilk and OpenMP 3.0 is that the latter can combine both types of parallelism, worksharing and tasks: for example, a programmer can choose to use OpenMP tasks to exploit the parallelism of an inner loop and the traditional worksharing construct to parallelize an outer loop.

1.3. Intel® Threading Building Blocks

Intel® Threading Building Blocks is a runtime-based parallel programming model for C++ code that uses threads. The main difference with other threading packages is that it enables the programmer to specify tasks instead of threads and the runtime library automatically schedules tasks onto threads in a way that makes efficient use of a multicore processor.

Another characteristic of TBB is that it focuses on the particular goal of parallelizing computationally intensive work, while this is not always true in general-purpose threading packages. TBB emphasizes data-parallel programming, enabling multiple threads to work on different parts of a collection enabling scalability to larger number of cores.

The programming model is based on template functions (parallel_for, parallel_reduce, etc.), where the user specifies the range of data to be accessed, how to partition the data, the task to be executed in each chunk.
1.4. SMPSs

SMP Superscalar (SMPSs) [16] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación), part of the STAR Superscalar family, which also includes Grid Supercalar and Cell Superscalar [18, 19]. While Grid Supercalar and Cell Superscalar address parallel software development for Grid environments and the Cell processor respectively, SMP Superscalar is aimed at "standard" (x86 and like) multicore processors and symmetric multiprocessor systems.

The principles of SMP Superscalar are similar to the ones of Cilk. Similarly to Cilk, the programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. Additionally, the programmer needs to specify the directionality of each parameter (input, output, inout). If the size of a parameter is missing in the C declaration (e.g., the parameter is passed by pointer), the programmer also needs to specify the size of the memory region affected by the function. Unlike Cilk, however, the programmer is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information of task parameters and their directionality.

Similarly to Cilk, the programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with pragma annotations to standard C99 code with calls to the supporting runtime library and compiles it using the platform native compiler.

At runtime the main thread creates worker threads, as many as necessary to fully utilize the system, and starts constructing the task graph (populating its ready list). Each worker thread maintains its own ready list and populates it while executing tasks. A thread consumes tasks from its own ready list in LIFO order. If that list is empty, the thread consumes tasks from the main ready list in FIFO order, and if that list is empty, the thread steals tasks from the ready lists of other threads in FIFO order.

The SMPSs scheduler attempts to exploit locality by scheduling dependent tasks to the same thread, such that output data is reused immediately. Also, in order to reduce dependencies, SMPSs runtime is capable of renaming data, leaving only the true dependencies, which is the same technique used by superscalar processors [20] and optimizing compilers [21].

The main difference between Cilk and SMPSs is that, while the former allows mainly for expression of nested parallelism, the latter handles computation expressed as an arbitrary DAG. Also, while Cilk requires the programmer to create the DAG by means of the spawn keyword, SMPSs creates the DAG automatically. Construction of the DAG does, however, introduce overhead, which is virtually inexistent in the Cilk environment.

1.5. Static Pipeline

The static pipeline scheduling presented here was originally implemented for dense matrix factorizations on the CELL processor [22, 23]. This technique is extremely simple and yet provides good locality of reference and load balance for regular computation, like dense matrix operations.

In this approach each task is uniquely identified by the \{m, n, k\} triple, which determines the type of operation and the location of tiles operated upon. Each core traverses its task space by applying a simple formula to the \{m, n, k\} triple, which takes into account the id of the core and the total number of cores in the system.
Task dependencies are tracked by a global progress table, where one element describes progress of computation for one tile of the input matrix. Each core looks up the table before executing each task to check for dependencies and stalls if dependencies are not satisfied. Each core updates the progress table after completion of each task. Access to the table does not require mutual exclusion (using, e.g., mutexes). The table is declared as volatile. Update is implemented by writing to an element. Dependency stall is implemented by busy-waiting on an element.

The use of a global progress table is a potential scalability bottleneck. It does not pose a problem, however, on small-scale multicore/SMP systems for small to medium matrix sizes. Many alternatives are possible. (Replicated progress tables were used on the CELL processor [22, 23]).

As further discussed in sections 3.3 and 4.3, this technique allows for pipelined execution of factorizations steps, which provides similar benefits to dynamic scheduling, namely, execution of the inefficient Level 2 BLAS operations in parallel with the efficient Level 3 BLAS operations.

The main disadvantage of the technique is potentially suboptimal scheduling, i.e., stalling in situations where work is available. Another obvious weakness of the static schedule is that it cannot accommodate dynamic operations, e.g., divide-and-conquer algorithms.

2. Related Work

Dynamic data-driven scheduling is an old concept and has been applied to dense linear operations for decades on various hardware systems. The earliest reference, that the authors are aware of, is the paper by Lord, Kowalik and Kumar [24]. A little later dynamic scheduling of LU and Cholesky factorizations were reported by Agarwal and Gustavson [25, 26]. Throughout the years dynamic scheduling of dense linear algebra operations has been used in numerous vendor library implementations such as ESSL, MKL and ACML (numerous references are available on the Web). In recent years the authors of this work have been investigating these ideas within the framework Parallel Linear Algebra for Multicore Architectures (PLASMA) at the University of Tennessee [27, 28, 29, 30]. Noteworthy is the implementation of sparse Cholesky factorization by Irony et al. using Cilk [31].

Seminal work leading to the tile QR algorithm presented here was done by Elmroth et al. [32, 33, 34]. Gunter et al. presented an "out-of-core" (out-of-memory) implementation [35], Buttari et al. an implementation for "standard" (x86 and alike) multicore processors [29, 30], and Kurzak et al. an implementation for the CELL processor [23].

Seminal work on performance-oriented data layouts for dense linear algebra was done by Gustavson et al. [36, 37] and Elmroth et al. [38] and was also investigated by Park et al. [39, 40].

3. Cholesky Factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used for the numerical solution of linear equations \(Ax = b\), where \(A\) is symmetric and positive definite. Such
systems arise often in physics applications, where $A$ is positive definite due to the nature of the modeled physical phenomenon. This happens frequently in numerical solutions of partial differential equations.

The Cholesky factorization of an $N \times N$ real symmetric positive definite matrix $A$ has the form

$$A = LL^T,$$

where $L$ is an $N \times N$ real lower triangular matrix with positive diagonal elements. In LAPACK the double precision algorithm is implemented by the DPOTRF routine. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: DSYRK, DPOTF2, DGEMM, DTRSM. Due to the symmetry, the matrix can be factorized either as upper triangular matrix or as lower triangular matrix. Here the lower triangular case is considered.

The algorithm can be expressed using either the top-looking version, the left-looking version of the right-looking version, the first being the most lazy algorithm (depth-first exploration of the task graph) and the last being the most aggressive algorithm (breadth-first exploration of the task graph). The left-looking variant is used here, with the exception of Cilk implementations, which favor the most aggressive right-looking variant.

The tile Cholesky algorithm is identical to the block Cholesky algorithm implemented in LAPACK, except for processing the matrix by tiles. Otherwise, the exact same operations are applied. The algorithm relies on four basic operations implemented by four computational kernels (Figure 1).

**Figure 1.** Tile operations in tile Cholesky factorization.

**DSYRK:** The kernel applies updates to a diagonal (lower triangular) tile $T$ of the input matrix, resulting from factorization of the tiles $A$ to the left of it. The operation is a symmetric rank-k update.
DPOTRF: The kernel performance the Cholesky factorization of a diagonal (lower triangular) tile \( T \) of the input matrix and overrides it with the final elements of the output matrix.

DGEMM: The operation applies updates to an off-diagonal tile \( C \) of the input matrix, resulting from factorization of the tiles to the left of it. The operation is a matrix multiplication.

DTRSM: The operation applies an update to an off-diagonal tile \( C \) of the input matrix, resulting from factorization of the diagonal tile above it and overrides it with the final elements of the output matrix. The operation is a triangular solve.

Figure 2 shows the pseudocode of the left-looking Cholesky factorization. Figure 3 shows the task graph of the tile Cholesky factorization of a \( 5 \times 5 \) tiles matrix. Although the code is as simple as four loops with three levels of nesting, the task graph is far from intuitive, even for a tiny size.

```plaintext
FOR k = 0..TILES-1
    FOR n = 0..k-1
        A(n)[k] = DSYRK(A(n)[n], A(n)[k])
        A(k)[k] = DPOTRF(A(k)[k])
    FOR m = k+1..TILES-1
        A(m)[k] = DGEMM(A(n)[n], A(m)[n], A(m)[k])
        A(m)[k] = DTRSM(A(k)[k], A(m)[k])
```

Figure 2. Pseudocode of tile Cholesky factorization (left-looking version).

Figure 3. Task graph of tile Cholesky factorization (\( 5 \times 5 \) tiles).
3.1. Cilk Implementation

Figure 4 presents implementation of Cholesky factorization in Cilk. The basic building blocks are the functions performing the tile operations. \textit{dsyrk()}, \textit{dtrsm()} and \textit{dgemm()} are implemented by calls to a single BLAS routine. \textit{dpotrf()} is implemented by a call to the LAPACK DPOTRF routine. The functions are declared using the \textit{cilk} keyword and then invoked using the \textit{spawn} keyword.

```cilk
for (k = 0; k < TILES; k++) {
    spawn dpotrf(A[k][k]);
    sync;
    for (m = k+1; m < TILES; m++)
        spawn dtrsm(A[k][k], A[m][k]);
    sync;
    for (m = k+1; m < TILES; m++)
        for (n = k+1; n < m; n++)
            spawn dgemm(A[k][n], A[m][n], A[m][k]);
    sync;
}
```

Figure 4. Cilk implementation of tile Cholesky factorization with 2D work assignment (right-looking version).

The input matrix is stored using the format referred to in literature as \textit{Square Block} (SB) format or \textit{Block Data Layout} (BDL). The latter name will be used here. In this arrangement, each function parameter is a pointer to a continuous block of memory, what greatly increases cache performance and virtually eliminates cache conflicts between different operations.

For implementation in Cilk the right-looking variant was chosen, where factorization of each panel is followed by an update to all the remaining submatrix. The code on Figure 4 presents a version, referred here as \textit{Cilk 2D}, where task scheduling is not constrained by data reuse considerations (There are no provisions for reuse of data between different tasks).

Each step of the factorization involves:

- factorization of the diagonal tile - spawning of the \textit{dpotrf()} task followed by a \textit{sync},
- applying triangular solves to the tiles below the diagonal tile - spawning of the \textit{dtrsm()} tasks in parallel followed by a \textit{sync},
- updating the tiles to the right of the panel - spawning of the \textit{dsyrk()} and \textit{dgemm()} tasks in parallel followed by a \textit{sync}.

It is not possible to further improve parallelism by pipelining the steps of the factorization. Nevertheless, most of the work can proceed in parallel and only the \textit{dpotrf()} task has to be executed sequentially.

Since the disregard for data reuse between tasks may adversely affect the algorithm’s performance, it is necessary to consider an implementation facilitating data reuse. One possible approach is processing of the tiles of the input matrix by columns. In this case, however, work is being dispatched in relatively big batches and load imbalance in each
step of the factorization will affect performance. A traditional remedy to this problem is the technique of *lookahead*, where update of step $N$ is applied in parallel with panel factorization of step $N + 1$. Figure 5 shows such implementation, referred here as *Cilk 1D*.

![Figure 5](image-url)

First, panel 0 is factorized, followed by a *sync*. Then updates to all the remaining columns are issued in parallel. Immediately after updating the first column, next panel factorization is spawned. The code synchronizes at each step, but panels are always overlapped with updates. This approach implements one-level lookahead (lookahead of depth one). Implementing more levels of lookahead would further complicate the code.

### 3.2. SMPSs Implementation

Figure 6 shows implementation using SMPSs. The functions implementing parallel tasks are designated with *pragma ccs task* annotations defining directionality of the parameters (input, output, inout). The parallel section of the code is designated with *pragma ccs start* and *pragma ccs finish* annotations. Inside the parallel section the algorithm is implemented using the canonical representation of four loops with three levels of nesting, which closely matches the pseudocode definition of Figure 2.

The SMPSs runtime system schedules tasks based on dependencies and attempts to maximize data reuse by following the parent-child links in the task graph when possible.

### 3.3. Static Pipeline Implementation

As already mentioned in section 1.5 the *static pipeline* implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms (volatile progress table and busy-waiting). Figure 7 shows the implementation.
#pragma css task input(A[NB][NB]) inout(T[NB][NB])
void dysr(k(double *A, double *T);

#pragma css task inout(T[NB][NB])
void dptrf(double *T);

#pragma css task input(A[NB][NB], B[NB][NB]) inout(C[NB][NB])
void dgemm(double *A, double *B, double *C);

#pragma css task inout(B[NB][NB])
void dtrsm(double *T, double *C);

#pragma css start
for (k = 0; k < TILES; k++) {
    for (n = 0; n < k; n++)
        dysr(A[k][n], A[k][k];
        dpotr(A[k][k]);
    for (m = k +1; m < TILES; m++) {
        for (n = 0; n < k; n++)
            dgemm(A[k][n], A[m][n], A[m][k]);
            dtrsm(A[k][k], A[m][k]);
    }
}
#pragma css finish

void dysr(double *A, double *T);
void dptrf(double *T);
void dgemm(double *A, double *B, double *C);
void dtrsm(double *T, double *C);

k = 0; m = my_core_id;
while (m >= TILES) {
    k++; m = m + TILES + k;
    if (n = 0)
        next_n = n; next_m = m; next_k = k;
    else {
        while (next_m > next_k) {
            next_m = next_m + cores_num;
        }
        if (next_m > next_k)
            next_n = n; next_m = next_m + TILES + next_k;
        else {
            next_m = next_m + cores_num;
            if (next_m > next_k)
                next_n = n; next_m = next_m - cores_num;
            else {
                while (next_m > next_k)
                    next_m = next_m - cores_num;
                if (next_m > next_k)
                    next_n = n; next_m = next_m + TILES + next_k;
                else {
                    next_m = next_m + cores_num;
                    if (next_m > next_k)
                        next_n = n; next_m = next_m - cores_num;
                    else {
                        next_m = next_m + cores_num;
                        next_m = next_m + TILES + next_k;
                    }
                }
            }
        }
    }
}

Figure 6. SMPSs implementation of tile Cholesky factorization (left-looking version).

Figure 7. Static pipeline implementation of tile Cholesky factorization (left-looking version).

The code implements the left-looking version of the factorization, where work is distributed by rows of tiles and steps of the factorization are pipelined. The first core that runs out of work in step \( N \) proceeds to factorization of the panel in step \( N + 1 \), following cores proceed to update in step \( N + 1 \), then to panel in step \( N + 2 \) and so on (Figure 8).
The code can be viewed as a parallel implementation of Cholesky factorization with one dimensional partitioning of work and lookahead, where lookahead of varying depth is implemented by processors which run out of work.

4. QR Factorization

The QR factorization (or QR decomposition) offers a numerically stable way of solving underdetermined and overdetermined systems of linear equations (least squares problems) and is also the basis for the QR algorithm for solving the eigenvalue problem.

The QR factorization of an \( m \times n \) real matrix \( A \) has the form

\[
A = QR,
\]

where \( Q \) is an \( M \times M \) real orthogonal matrix and \( R \) is an \( M \times N \) real upper triangular matrix. The traditional algorithm for QR factorization applies a series of elementary Householder matrices of the general form

\[
H = I - \tau vv^T,
\]

where \( v \) is a column reflector and \( \tau \) is a scaling factor. In the block form of the algorithm a product of \( NB \) elementary Householder matrices is represented in the form

\[
H_1H_2\ldots H_{NB} = I - VTV^T,
\]

where \( V \) is an \( N \times N \) real matrix those columns are the individual vectors \( v \), and \( T \) is an \( NB \times NB \) real upper triangular matrix [41, 42]. In LAPACK the double precision algorithm is implemented by the DGEQRF routine.

Here a derivative of the block algorithm is used called the tile QR factorization. The ideas behind the tile QR factorization are very well known. The tile QR factorization was initially developed to produce a high-performance "out-of-memory" implementation (typically referred to as "out-of-core") [35] and, more recently, to produce high performance implementation on "standard" (x86 and alike) multicore processors [29, 30] and on the CELL processor [23].
The algorithm is based on the idea of annihilating matrix elements by square tiles instead of rectangular panels (block columns). The algorithm produces the same $R$ factor as the classic algorithm, e.g., the implementation in the LAPACK library (elements may differ in sign). However, a different set of Householder reflectors is produced and a different procedure is required to build the $Q$ matrix. Whether the $Q$ matrix is actually needed depends on the application. The tile QR algorithm relies on four basic operations implemented by four computational kernels (Figure 9).

**Figure 9.** Tile operations in tile QR factorization.

**DGEQRT:** The kernel performs the QR factorization of a diagonal tile of the input matrix and produces an upper triangular matrix $R$ and a unit lower triangular matrix $V$ containing the Householder reflectors. The kernel also produces the upper triangular matrix $T$ as defined by the compact WY technique for accumulating Householder reflectors [41, 42]. The $R$ factor overrides the upper triangular portion of the input and the reflectors override the lower triangular portion of the input. The $T$ matrix is stored separately.

**DTSQRT:** The kernel performs the QR factorization of a matrix built by coupling an $R$ factor, produced by DGEQRT or a previous call to DTSQRT, with a tile below the diagonal tile. The kernel produces an updated $R$ factor, a square matrix $V$ containing the Householder reflectors and the matrix $T$ resulting from accumulating the reflectors $V$. The new $R$ factor overrides the old $R$ factor. The block of reflectors overrides the square tile of the input matrix. The $T$ matrix is stored separately.

**DLARFB:** The kernel applies the reflectors calculated by DGEQRT to a tile to the right of the diagonal tile, using the reflectors $V$ along with the matrix $T$.

**DSSRFB:** The kernel applies the reflectors calculated by DTSQRT to two tiles to the right of the tiles factorized by DTSQRT, using the reflectors $V$ and the matrix $T$ produced by DTSQRT.

Naive implementation, where the full $T$ matrix is built, results in 25% more floating point operations than the standard algorithm. In order to minimize this overhead, the idea of *inner-blocking* is used, where the $T$ matrix has sparse (block-diagonal) structure (Figure 10) [32, 33, 34].

Figure 11 shows the pseudocode of tile QR factorization. Figure 12 shows the task graph of the tile QR factorization for a matrix of $5 \times 5$ tiles. Orders of magnitude larger
matrices are used in practice. This example only serves the purpose of showing the complexity of the task graph, which is noticeably higher than that of Cholesky factorization.

4.1. Cilk Implementation

The task graph of the tile QR factorization has a much denser net of dependencies than the Cholesky factorization. Unlike for Cholesky the tasks factorizing the panel are not independent and have to be serialized and the tasks applying the update have to follow the same order. The order can be arbitrary. Here top-down order is used.

Figure 13 shows the first Cilk implementation, referred to as Cilk 2D, which already requires the use of lookahead to achieve performance. The basic building block are the functions performing the tile operations. Unlike for Cholesky, none of them is a simple call to BLAS or LAPACK. Due to the use of inner-blocking the kernels consist of loop nests containing a number of BLAS and LAPACK calls (currently coded in FORTRAN 77).

The factorization proceeds in the following steps:

- Initially the first diagonal tile is factorized - spawning of the dgeqrt() task followed by a sync. Then the main loop follows with the remaining steps.
Tiles to the right of the diagonal tile are updated in parallel with factorization of the tile immediately below the diagonal tile - spawning of the dlarfb() tasks and the dsqrt() task followed by a sync.

Updates are applied to the tiles right from the panel - spawning of the dssrfb() tasks by rows of tiles (sync following each row). The last dssrfb() task in a row spawns the dsqrt() task in the next row. The last dssrfb() task in the last row
spawns the \texttt{dgeqrt()} task in the next step of the factorization.

Although lookahead is used and factorization of the panel is, to some extent, overlapped with applying the update, tasks are being dispatched in smaller batches, what severely limits opportunities for scheduling.

The second possibility is to process the tiles of the input matrix by columns, the same as was done for Cholesky. Actually, it is much more natural to do it in the case of QR, where work within a column has to be serialized. Load imbalance comes into picture again and lookahead is the remedy. Figure 14 shows the implementation, referred to as Cilk 1D.

\begin{verbatim}
void dgeqrt(double *RV1, double *T);
void dsqrt(double *R, double *Q2, double *T1);
void dlartf(double *V1, double *T, double *C1);
void dssrft(double *V2, double *T, double *C1, double *C2);

cilk void qr_panel(int k)
{
    int m;
    dgeqrt(A[k][k], T[k][k]);
    for (m = k+1; m < TILES; m++)
        dsqrt(A[k][k], A[m][k], T[m][k]);
}
cilk void qr_update(int n, int k)
{
    int m;
    dlartf(A[k][k], T[k][k], A[k][n]);
    for (m = k+1; m < TILES; m++)
        dssrft(A[m][k], T[m][k], A[k][n], A[m][n]);
    if (n == k+1)
        spawn qr_panel(k+1);
}

spawn qr_panel(0);
sync;
for (k = 0; k < TILES; k++)
{
    for (n = k+1; n < TILES; n++)
        spawn qr_update(n, k);
    sync;
}
\end{verbatim}

**Figure 14.** Cilk implementation of tile QR factorization with 1D work assignment and lookahead.

The implementation follows closely the Cilk 1D version of Cholesky. First, panel 0 is factorized, followed by a \texttt{sync}. Then updates to all the remaining columns are issued in parallel. Immediately after updating the first column, next panel factorization is spawned. The code synchronizes at each step, but panels are always overlapped with updates. This approach implements one-level lookahead (lookahead of depth one). Implementing more levels of lookahead would further complicate the code.

4.2. **SMPSs Implementation**

Figure 15 shows implementation using SMPSs, which follows closely the one for Cholesky. The functions implementing parallel tasks are designated with \texttt{#pragma ccs task} annotations defining directionality of the parameters (input, output, inout). The parallel section of the code is designated with \texttt{#pragma ccs start} and \texttt{#pragma ccs finish} annotations. Inside the parallel section the algorithm is implemented using the canonical representation of four loops with three levels of nesting, which closely matches the pseudocode definition of Figure 11.
Figure 15. SMPSs implementation of tile QR factorization.

The SMPSs runtime system schedules tasks based on dependencies and attempts to maximize data reuse by following the parent-child links in the task graph when possible.

There is a caveat here, however. V1 is an input parameter of task `dlarfb()`. It also is an inout parameter of task `dtsqrt()`. However, `dlarfb()` only reads the lower triangular portion of the tile, while `dtsqrt()` only updates the upper triangular portion of the tile. Since in both cases the tile is passed to the functions by the pointer to the upper left corner of the tile, SMPSs sees a false dependency. As a result, the execution of the `dlarfb()` tasks in a given step will be stalled until all the `dtsqrt()` tasks complete, despite the fact that both types of tasks can be scheduled in parallel as soon as the `dgeqrt()` task completes. Figure 16 shows conceptually the change that needs to be done.

Currently SMPSs is not capable of recognizing accesses to triangular matrices. There are however multiple ways to enforce the correct behavior. The simplest method, in this case, is to drop dependency check on the V1 parameter of the `dlarfb()` function by declaring it as `volatile*`. Correct dependency will be enforced between the `dgeqrt()` task and the `dlarfb()` tasks through the T parameter. This implementation is further referred to as `SMPSs*`.

4.3. Static Pipeline Implementation

The static pipeline implementation for QR is very close to the one for Cholesky. As already mentioned in section 1.5 the static pipeline implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms (`volatile` progress table and busy-waiting). Figure 17 shows the implementation.

The code implements the right-looking version of the factorization, where work is distributed by columns of tiles and steps of the factorization are pipelined. The first core that runs out of work in step $N$ proceeds to factorization of the panel in step $N + 1$, following cores proceed to update in step $N + 1$, then to panel in step $N + 2$ and so on (Figure 18).
The goal here is to clearly illustrate differences in scheduling by the different implementations of tile QR factorization with improved dependency resolution for diagonal tiles.

Figure 16. SMPSs implementation of tile QR factorization with improved dependency resolution for diagonal tiles.

The code can be viewed as a parallel implementation of tile QR factorization with one dimensional partitioning of work and lookahead, where lookahead of varying depth is implemented by processors which run out of work.

5. Results and Discussion

Figure 19 shows execution traces of all the implementations of Cholesky factorization. The figure shows a small run \((11 \times 11 \text{ tiles, } 1320 \times 1320 \text{ elements})\) on a small number of cores (five). The goal here is to clearly illustrate differences in scheduling by the different approaches.

The Cilk 1D implementation performs the worst. The 1D partitioning of work causes a disastrous load imbalance in each step of the factorization. Despite the lookahead, panel execution is very poorly overlapped with the update, in part due to the triangular shape of the updated submatrix and quickly diminishing amount of work in the update phase.

The Cilk 2D implementation performs much better by scheduling the \texttt{dtrsm()} operations in the panel in parallel. Also, scheduling the \texttt{dsyrk()} and \texttt{dgemm()} tasks in the update phase without constraints minimizes load imbalance. The only serial task, \texttt{dpotrf()}, does not cause disastrous performance losses.

Yet better is the SMPSs implementation, where tasks are continuously scheduled without gaps until the very end of the factorization, where the natural load imbalance occurs. Data reuse is clearly visible through clusters of \texttt{dsyrk()} tasks. The only inefficiency affecting the performance is the non-negligible startup cost.

The static pipeline schedule is clearly superior. It is virtually free of dependency stalls until the very end of the factorization, maximizes data reuse and is free of startup overheads.

Figure 20 shows execution traces of all the implementations of QR factorization. The same as for Cholesky, the figure shows a small run \((9 \times 9 \text{ tiles, } 1296 \times 1296 \text{ elements})\) on a
void dgeqrt(double *R1, double *T1);
void dsqrt(double *R, double *V2, double *T1);
void dsrftb(double *V1, double *T, double *C1);
void dsrftb(double *V2, double *T, double *C1, double *C2);

k = 0; n = my_core_id;
while (n >= TIMES) {
  k++; n = m TIMES + k;
}

while (k < TIMES && n < TIMES) {
  next_n = n; next_m = m; next_k = k;
  next_m += coresize
  if (next_m == TIMES) {
    next_n += coresize
  } while (next_n >= TIMES && next_k < TIMES) {
    next_k += k; next_n = next_n TIMES + next_k;
  } next_m = next_k;
}

while_progress(k) &= k-1;
dgeqrt(k, k, k, T(k, k));
}

else{
  while_progress(m) &= k-1;
dsqrt(A(k, k), A(m, k), T(m));
  progress[m] = k;
}

} else {
if (m = k) {
  while_progress(k) &= k;
  while_progress(n) &= k-1;
drftb(A(k, k), T(k), A(k))
}

else {
  while_progress(m) &= k;  
  while_progress(m) &= k-1;
  dsrftb(A(m, k), T(m), A(k), A(m, k)),
  progress[m] = k;
}

} n = next_n; m = next_m; k = next_k;

Figure 17. Static pipeline implementation of tile QR factorization.

small number of cores (five). Once again, the goal here is to clearly illustrate differences in scheduling by the different approaches.

The situation looks a bit different for the tile QR factorization. Unlike for Cholesky, the fine-grain Cilk 2D implementation performs poorest, which is mostly due to the dispatch of work in small batches. Although the tasks of panel factorization (dgeqrt(), dsqrt()) are overlapped with the task of the update (dsrftb(), dsrftb()), synchronization after each row, and related load imbalance, contribute big number of gaps in the trace.

The Cilk 1D version performs better. Although the number of gaps is still significant, mostly due to 1D partitioning and related load imbalance, overall this implementation looses less time due to dependency stalls.

Figure 18. Work assignment in the static pipeline implementation of tile QR factorization.
Interestingly the initial SMPSs implementation produces almost identical schedule to the Cilk 1D version. One difference is the startup cost at the beginning of the SMPSs trace, the other is the better schedule at the end. The overall performance difference is minimal.

The SMPSs* implementation delivers a big jump in performance, due to dramatic improvement in the schedule. Similarly to the static pipeline schedule for Cholesky, the one for QR is virtually free of dependency stalls until the very end of the factorization, data reuse is clearly seen and there are no startup overheads.

Figure 21 shows performance in Gflop/s of different implementations. Results were collected on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system running Linux kernel 2.6.18. Cilk 5.4.6 was used, SMPSs 2.0 and MKL 10.0.1. In each case matrices were stored in Block Data Layout and memory was allocated using huge TLB pages.

Figure 19. Execution traces of tile Cholesky factorization in double precision on five cores of a 2.4 GHz Intel Tigerton system. Matrix size $N = 1320$, tile size $NB = 120$.

Figure 21 shows performance for the Cholesky factorization, where Cilk implementations provide mediocre performance, SMPSs provides much better performance and static pipeline provides performance clearly superior to other implementations.

Figure 22 shows performance for the QR factorization. The situation is a little different here. Performance of Cilk implementations is still the poorest and the performance of the static pipeline is still superior. However, performance of the initial SMPSs implementation is only marginally better that Cilk 1D, while performance of the improved SMPSs* implementation is only marginally worse that static pipeline.

Relatively better performance of SMPSs for the QR factorization versus the Cholesky factorization can be explained by the fact that the QR factorization is four times more expensive in terms of floating point operations, what diminishes the impact of various overheads for smaller size problems.
Figure 20. Execution traces of tile QR factorization in double precision on five cores of a 2.4 GHz Intel Tigerton system. Matrix size $N = 1296$, tile size $NB = 144$, inner block size $IB = 48$.

Figure 21. Performance of tile Cholesky factorization in double precision on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. Tile size $NB = 120$.

6. Conclusions

In this work, suitability of emerging multicore programming frameworks was analyzed for implementing modern formulations of classic dense linear algebra algorithms, tile Cholesky and tile QR factorizations. These workloads are represented by large task graphs with compute-intensive tasks interconnected with a very dense and complex net of dependencies.
Figure 22. Performance of tile QR factorization in double precision on a 2.4 GHz quad-socket quad-core (16 cores total) Intel Tigerton system. Tile size $NB = 144$, inner block size $IB = 48$.

For the workloads under investigation, the conducted experiments show clear advantage of the model, where automatic parallelization is based on construction of arbitrary DAGs. SMPSs provides much higher level of automation than Cilk and similar frameworks, requiring only minimal programmer’s intervention and basically leaving the programmer oblivious to any aspects of parallelization. At the same time it delivers superior performance through more flexible scheduling of operations.

SMPSs still looses to hand-written code for very regular compute-intensive workloads investigated here. The gap is likely to decrease, however, with improved runtime implementations. Ultimately, it may have to be accepted as the price for automation.

7. Future Directions

Parallel programing based on the idea of representing the computation as a task graph and dynamic data-driven execution of tasks shows clear advantages for multicore processors and multi-socket shared-memory systems of such processors. One of the most interesting questions is the applicability of the model to large scale distributed-memory systems.

References


Algorithms and scheduling techniques for clusters and grids

Anne BENOIT \textsuperscript{2,4}, Loris MARCHAL \textsuperscript{1,4}, Yves ROBERT \textsuperscript{2,4}, and Frédéric VIVIEN \textsuperscript{3,4}

\textsuperscript{1} CNRS \textsuperscript{2} ENS Lyon \textsuperscript{3} INRIA \textsuperscript{4} Université de Lyon


Abstract The main objective of this chapter is to show the need for algorithmic and scheduling techniques. Even if resources at our disposal would become abundant and cheap, not to say unlimited and free (a perspective that is not granted), we would still need to assign the right task to the right device. We give several examples of such situations where careful resource selection and allocation are mandatory. Finally we outline some important algorithmic challenges that need be addressed in the future.

Keywords. Algorithm design, scheduling techniques.

1. Introduction

Already in the former century, scheduling was sometimes considered as a minor, and more or less useless, activity. Today the question is raised much strongly. With over a billions of (mostly idle) computers in the world, all interconnected by these (partially empty) network pipes, the resources at our disposal become abundant and cheap, not to say unlimited and free. Well, at least there is a chance for this dream to become true. Who would then need a complicated scheduling algorithm while a greedy resource allocation is likely to do the job? Demand-driven approaches like \textit{first-come first-serve} or \textit{round-robin} will perform extremely well in most situations. In short, \textit{who needs a scheduler with infinite resources handy and ready}?

The aim of this chapter is to demonstrate that algorithm design and scheduling techniques remain fully useful, not to say unavoidable, despite the advent of ubiquitous computing facilities. The more resources at our disposal, the more difficult the art of selecting which ones to enroll in the execution, and of mapping the right task onto the right machine. The resource selection and mapping processes turn out difficult because the algorithm used to derive the optimal solution may be counter-intuitive. For instance, considering the computational speed of candidate processors is sometimes irrelevant (see Section 3.1). We show that performing no resource selection, or using a poor one, can lead to very low performance. Furthermore, we should rather use the huge computing power at hand in order to solve larger problems more efficiently rather than wasting resources. This is especially true as energy consumption becomes a more and more important problem.
We use simple examples such as bag-of-tasks applications or matrix product as case studies for our demonstrations. Then we address more challenging applications, which require to simultaneously deal with several (usually conflicting) optimization criteria.

Before moving to the examples and case studies, we clarify our definition of the word *scheduling*, because it has different meanings in the literature. In this chapter we deal with what is usually called *static scheduling*, an activity which starts with a set of tasks (potentially organized as a precedence graph, or DAG), and with a target computing platform, as input, and consists in mapping the former on the latter with the goal to optimize some objective function (often the total execution time, or *makespan*). Static schedulers need a reasonably good knowledge of the application parameters. More precisely, the structure of the DAG, and estimations of node and edge weights (which correspond to computation costs and communication volumes respectively) are fed into the scheduler.

This is different from *dynamic scheduling*, or better-named *demand-driven resource allocation*, which consists in mapping jobs onto shared computational resources. Typically, very little is known about the jobs, maybe rough estimates of their execution times in some cases, and nothing is known in advance about their incoming rate. In such situations, there is not much else to do than assigning new loads to currently idle resources, satisfying requests with a simple First-Come, First-Served policy.

The terms *static* and *dynamic* are somewhat misleading, because a scheduler can (dynamically) take new decisions on the fly, based upon newly acquired information on application and platform parameters. We refer to *scheduling* as the activity of designing algorithms and heuristics (e.g., a list schedule) in order to deploy an application onto a platform. On the contrary, a *demand-driven* approach is a system-oriented approach where resources are allocated to incoming requests upon demand. Of course this is an overly simplified classification and there is a continuum. When several applications (rather than one) are simultaneously deployed by a single user on a platform shared by many other users (rather than on a dedicated platform), the difference between both approaches narrows. To our view, the difference goes beyond off-line versus on-line, or compile-time versus run-time. Basically, the more we know about what we need to schedule, the more refined the decisions that the scheduler can take.

This clarification being made, we move to the contents of this chapter. In Section 2, we discuss the importance of realistic communication models, and we introduce the one-port and bounded multi-port models that we use in all examples. In Section 3 we present the first case-study, that of bag-of-tasks applications (i.e., collections of identical tasks). Despite the utmost simplicity of such applications, we mathematically assess the importance of resource selection and load assignment strategies. Next we deal with the second case study, namely matrix product under memory constraints, in Section 4. Here we show that strategies that minimize communication volume are the key to effective resource utilization.

Both case studies avoid the complexity of makespan minimization by addressing a simpler but related optimization objective: the throughput in Section 3 and the communication volume in Section 4. In real life, problems should be stated with several (conflicting) objectives rather than just one. Such multi-criteria problems are outlined in Section 5. Finally we state some final remarks in Section 6.
2. Communication models

Communications often play a major role in application performance (see for instance Section 4), hence a need for accurate communication models. In this section, we describe the standard communication model used in the scheduling literature, and we explain why it is completely inaccurate for current computing platforms. Then we introduce two more realistic models, which we believe represent a much better trade-off between realism and tractability.

2.1. The macro-dataflow model

Distributed-memory parallel computing platforms pose many challenges to the algorithm designer and the programmer. An obvious factor contributing to this complexity is the need for network communications, whose performance is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms. Older parallel computers used a store-and-forward approach to communicate messages, which was not efficient but simple to understand and to model. In this context, the time for sending a message from a processor \( p \) to a processor \( p' \) is

\[
c(p, p') = \text{dist}(p, p') \times (L + s/b),
\]

where \( s \) is the length of the message, \( \text{dist}(p, p') \) is the distance between \( p \) and \( p' \) in number of hops, \( L \) is the communication start-up cost, and \( b \) is the steady-state bandwidth. In modern computers, messages are split into packets that are dynamically routed between processors, possibly using different paths. Messages can be routed efficiently if there are no contentions on the communication links (or “hot spots”). The distance between communicating processors is no longer the single most important factor for communication performance. In fact, if several processors are to exchange data simultaneously, then the more structured the communication patterns, the more efficient they are, making the role of locality on performance at best indirect.

In light of the complexity of performance modeling for network communications, the vast majority of scheduling works and results are for a very simple model, which is as follows. If a task \( T \) communicates data to a successor task \( T' \), the cost is modeled as

\[
\text{cost}(T, T') = \begin{cases} 
0 & \text{if } \text{alloc}(T) = \text{alloc}(T') \\
\ c(T, T') & \text{otherwise,}
\end{cases}
\]

where \( \text{alloc}(T) \) denotes the processor that executes task \( T \), and \( c(T, T') \) is defined by the application specification. The above model states that the time for communication between two tasks running on the same processor is negligible. The model also assumes that the processors are part of a fully connected clique. This so-called macro-dataflow model makes three main assumptions: (i) communication can occur as soon as data are available; (ii) the communication network is homogeneous; and (iii) there is no contention for network links. Assumption (i) is reasonable as communications can overlap with computations in most modern computers. Assumption (ii) is fair on a single cluster of workstations, but inaccurate for large-scale platforms. Assumption (iii) is much more questionable. Indeed, there is no physical device capable of sending, say, 1,000 messages to 1,000 distinct processors, at the same speed as if there were a single message. In the worst case, it would take 1,000 times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (iii) amounts to assuming infinite network resources! Neverthe-
less, this assumption is omnipresent in the traditional scheduling literature. Perhaps was it the price to pay to derive tractable mathematical results on makespan minimization?

Our conviction is that we need to turn to more realistic communication models when modeling concurrent communications. We outline two such models, that account for the interference between concurrent communications.

2.2. The bounded multi-port model

Assuming an application that runs threads on, say, a node that uses multicore technology, the network link could be shared by several incoming and outgoing communications. Therefore, the sum of the bandwidths allotted by the operating system to all communications cannot exceed the bandwidth of the network card. The bounded multi-port model proposed by Hong and Prasanna [22] assesses that an unbounded number of communications can thus take place simultaneously, provided that they share the total available bandwidth. We point out that recent multi-threaded communication libraries such as MPICH2 [24] now allow for initiating multiple concurrent send and receive operations, thereby providing practical realizations of the multi-port model.

Note that with this model there is no degradation of the aggregate throughput. Such a behavior is typical for protocols with efficient congestion control mechanisms (e.g., TCP). Note, however, that this model does not express how the bandwidth is shared among the concurrent communications. It is generally assumed in this model that the application is allowed to define the bandwidth allotted to each communication. In other words, bandwidth sharing is performed by the application and not by the operating system. While technology exists to achieve application-level bandwidth sharing, it is not the standard way in which networks and operating systems operate.

2.3. The one-port model

A radical option is simply to forbid concurrent communications at each node. In the one-port model, a node can either send data or receive data, but not simultaneously. This model is thus very pessimistic as real-world platforms can achieve some concurrency of communication. On the other hand, it is straightforward to design algorithms that follow this model and thus to determine their performance a priori.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It is used by Bhat et al. [9,10] for fixed-sized messages. They advocate its use because “current hardware and software do not easily enable multiple messages to be transmitted simultaneously.” Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations “are eventually serialized by the single hardware port to the network.” Experimental evidence of this fact has been related by Saif and Parashar [30], who report that asynchronous sends become serialized as soon as message sizes exceed a few tens of kilobytes. (Their results hold for two popular implementations of the MPI message-passing standard, MPICH on Linux clusters and IBM MPI on the SP2.)

There are more complicated models such as those that deal with bandwidth sharing protocols [25,26]. Such models are very interesting for performance evaluation purposes, but they almost always prove too complicated for algorithm design purposes. For this reason, we prefer to deal with the bounded multi-port or the one-port model. As
stated above, we believe that these models represent a good trade-off between realism and tractability, and we use them for all the examples and case studies of this chapter.

3. Case study: bag-of-tasks applications

In this section we study the deployment of BOINC-like applications [14] under the previous one-port and bounded multi-port models. We tackle three increasingly difficult problems:

- We start with the simplest problem, that of scheduling a single bag-of-tasks application made up of a large number of same-size tasks onto a master-worker platform. It turns out that the choice of the model has a dramatic impact on the solution: resource selection is mandatory under the one-port model, but can be alleviated with the bounded multi-port model.

- Next we proceed with several bag-of-tasks applications onto the same simple master-worker platform. The choice of the model makes no longer a difference: for both of them, a sophisticated scheduling algorithm is needed to ensure good performance.

- Finally we discuss both previous problems on general platforms (instead of simple master-worker platforms). Everything becomes quite complicated!

For all problems, we need to abandon the hope to minimize the total execution time (or makespan), as for most scheduling problems, makespan minimization is NP-hard, even for a single bag-of-task application onto a tree platform [19]. A modern approach to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective. Instead of aiming at the absolute minimization of the execution time, why not consider asymptotic optimality? Often, the motivation for deploying an application on a parallel platform is that the number of tasks is very large. In this case, the optimal execution time with the optimal schedule may be very large and a small deviation from it is likely acceptable. To state this informally: if there is a nice (e.g., polynomial) way to derive, say, a schedule whose length is two hours and three minutes, as opposed to an optimal schedule that would run for only two hours, we would be satisfied.

Steady-state scheduling—an approach pioneered by Bertsimas and Gamarnik [8]—allows one to relax the scheduling problem in many ways. The costs of the initialization and clean-up phases are neglected. The initial integer formulation is replaced by a continuous, i.e., rational, formulation. The precise scheduling of computations and communications is not required, or at least not before the optimal schedule is outlined. The main idea is to characterize the activity of each resource during each time unit: which (rational) fraction of time is spent computing, which is spent receiving or sending to which neighbor. Such activity variables are gathered into a linear program, which includes conservation laws that characterize the global behavior of the system. The actual schedule then arises naturally from these quantities and can be proved to be asymptotically optimal. In the following, we illustrate steady-state scheduling techniques first with a single bag-of-tasks application (Section 3.1) and then with several ones (Section 3.2).
3.1. One bag-of-tasks application

In this section, we target a single bag-of-tasks application on a simple heterogeneous star-shaped platforms. The master $M$ initially holds a large collection of atomic tasks. Refer to Figure 1 for notations:

- The master $M$ sends tasks to workers without preemptions. It sends these tasks \textit{sequentially} (one-port model) or \textit{in parallel} (bounded multi-port model).
- There is full computation/communication overlap on each worker.
- A task consists of an input file of size $\delta$ (in Bytes), and a computation job of size $w$ (in Flops).
- Worker $P_i$ has a communication bandwidth $b_i$: it receives a task in $\delta/b_i$ time units.
- Worker $P_i$ has a computation speed $s_i$: it processes a task in $w/s_i$ time units.
- The master $M$ does not compute any task (but a master with computation speed $s_0$ can be simulated as a worker with the same computation speed and infinite bandwidth).

When dealing with a single bag-of-tasks application, we assume that $\delta = w = 1$ without loss of generality (processor speeds and bandwidths can be scaled).

The optimal steady-state is defined as follows: for each worker, determine the fraction of time spent computing tasks, and the fraction of time spent receiving tasks; for the master, determine the fraction of time spent communicating along each communication link. The objective is to maximize the (average) number of tasks processed per time unit. Formally, after a start-up phase, we want the resources to operate in a periodic mode, with worker $P_i$ executing $\alpha_i$ tasks per time unit. We point out that $\alpha_i$ is a rational number, not an integer one, so that there remains some work to reconstruct a feasible schedule, i.e., with an integer number of tasks \cite{4}.

3.1.1. One-port model

First we express the constraints for computations: $P_i$ must compute $\alpha_i$ tasks within one time unit, thus we must have $s_i \geq \alpha_i$, and

$$\alpha_i/s_i \leq 1.$$  \hspace{1cm} (1)

As for communications, the master $M$ sends tasks sequentially to the workers, and it must send $\alpha_i$ tasks per time unit along the link to $P_i$. Thus, by summing all communication times we obtain
Finally, the objective is to maximize the throughput, namely,

$$\rho = \sum_{i=1}^{p} \alpha_i$$.

Altogether, we have a linear programming problem with rational unknowns:

**MAXIMIZE** $\rho$,

**SUBJECT TO**

$$\rho = \sum_{i=1}^{p} \alpha_i$$ (i)

$$\sum_{i=1}^{p} \alpha_i / b_i \leq 1$$ (ii)

$$\forall i, \alpha_i / s_i \leq 1$$ (iii)

$$\forall i, \alpha_i \geq 0$$ (iv)

It turns out that the linear program is so simple that it can be solved analytically. Indeed it is a fractional knapsack problem [17] with value-to-cost ratio $b_i$. We should start with the “item” (worker) of the largest ratio, i.e., the largest $b_i$, and take (assign) as many tasks as we can, i.e., $\min (b_i, s_i)$. Here is the detailed procedure:

1. Sort the workers by increasing communication times. Re-number them so that $b_1 \geq b_2 \ldots \geq b_k$.
2. Let $q$ be the largest index so that $\sum_{i=1}^{q} \frac{s_i}{b_i} \leq 1$. Workers $P_1$ to $P_q$ will be fully active (and each of them will execute $s_i$ tasks per time unit). If $q < p$, let $\varepsilon = 1 - \sum_{i=1}^{q} \frac{s_i}{b_i}$, otherwise let $\varepsilon = 0$. Worker $P_{q+1}$ (if it exists) will be only partially active, and will execute $\varepsilon \cdot b_{q+1}$ tasks per time unit.
3. Workers $P_{q+2}$ to $P_p$ (if they exist) are discarded; they will not participate in the computation.
4. The optimal throughput is then

$$\rho = \sum_{i=1}^{q} s_i + \varepsilon \cdot b_{q+1}$$.

When $q = p$ the result is expected. It basically says that workers can be fed with tasks fast enough so that they are all kept computing steadily. However, if $q < p$, the result is surprising. Indeed, if the communication bandwidth is limited, some workers will partially starve. In the optimal solution these partially starved workers are those with slow communication rates, regardless of their processing speeds. In other words, a slow processor with a fast communication link is to be preferred to a fast processor with a slow communication link. This optimal strategy is often called bandwidth-centric because it delegates work to the fastest communicating workers, regardless of their computing speeds. Of course, slow computing workers will not contribute much to the overall throughput.
Consider the example shown in Figure 2. Workers are sorted by non-increasing $b_i$. We see that $\frac{s_1}{b_1} + \frac{s_2}{b_2} = \frac{15}{20} \leq 1$ and that $\frac{s_1}{b_1} + \frac{s_2}{b_2} + \frac{s_3}{b_3} = \frac{65}{20} > 1$, so that $q = 2$ and $\varepsilon = \frac{1}{4}$ in the previous formula. Therefore, $P_1$ and $P_2$ will be fully active, contributing $\alpha_1 + \alpha_2 = s_1 + s_2 = 11$ tasks per time unit. $P_3$ will only be partially active, contributing $\alpha_3 = \varepsilon \cdot b_{q+1} = 1$ task per time unit. $P_4$ and $P_5$ will be discarded. The optimal throughput is $\rho = 7 + 4 + 1 = 12$. Figure 2(b) shows that 12 tasks are computed every time unit.

It is important to point out that if we had used a purely greedy (demand-driven) strategy, we would have reached a much lower throughput. Indeed, one can show that under a demand-driven strategy the master serves the workers in a round-robin fashion, and only 5 tasks are executed every $\frac{1}{20} + \frac{1}{10} + \frac{1}{4} + \frac{1}{2} + 1 = \frac{10}{10}$ time units, therefore achieving a throughput of only $\rho = \frac{10}{19} \approx 0.53$. The conclusion is that even when resources are cheap and abundant, resource selection is key to performance. (Here the best solution only uses the three slowest computing processors!)

The good news is that the actual periodic schedule can easily be constructed from the linear program, and that this schedule is asymptotically optimal. See [4] for details.

3.1.2. Bounded multi-port model

How can we solve the same problem using the bounded multi-port model instead of the one-port model? Refer to the one-port linear program again. Because messages can now be sent in parallel, we replace Equation (ii) by

$$\forall i, \quad \frac{\alpha_i}{b_i} \leq 1,$$

which states that the bandwidth of the link from $M$ to $P_i$ is not exceeded. We also have to enforce a global bound related to the bandwidth $B$ of the master’s network card:

$$\sum_{i=1}^{p} \frac{\alpha_i}{B} \leq 1.$$
Replacing Equation (ii) by both Equations (ii-a) and (ii-b) is all that is needed to change to the bounded multi-port model.

However, this modification has a dramatic impact on the solution and on the scheduler. Resource selection is not needed any longer. If we enroll all (or sufficiently many) available resources and feed each of them using a pure demand-driven basis (thereby enforcing that $\alpha_i \leq \min(s_i, b_i)$), we end up reaching the maximum throughput $\rho_{opt} = \min \left( B, \sum_{i=1}^{n} \min(s_i, b_i) \right)$ dictated by the master’s outgoing communication capacity.

This seems contradictory with our initial claim. Is the complexity an artifact of the one-port model? We will see in the following that even in the “simple” multi-port model, static knowledge is required to efficiently schedule several applications.

3.2. Several bag-of-tasks applications

We now consider that a single scheduler has to cope with tasks belonging to several applications. There are $K$ applications $(A_1, \ldots, A_K)$, and each application consists of a large number of same-size tasks, to be executed on the same master-worker platform. Some new notations are needed:

- $\delta_k$ is the size (in Bytes) of an input file for application $A_k$; processor $P_i$ receives a task of $A_k$ in $\frac{\delta_k}{b_i}$ time units.
- $w_k$ is the size (in Flops) of a task for application $A_k$; processor $P_i$ executes a task of $A_k$ in $\frac{w_k}{s_i}$ time units.

When dealing with several applications in steady-state mode, $\alpha_{ki}$ denotes the local throughput of application $A_k$ on processor $P_i$. In other words, processor $P_i$ executes $\alpha_{ki}$ tasks of applications $A_k$ during one time unit. As previously, $\alpha_{ki}$ might be a rational number. The total throughput $\rho^k$ of an application $A_k$ is then given by $\rho^k = \sum_{i=1}^{p} \alpha_{ki}$.

Since we have several applications to schedule on the same platform, we have to modify the objective to take all applications into account. We assume that some applications may be more important than others. Each application $A_k$ is provided with a priority $\pi_k$, so that if $\pi_k = 2\pi_{k'}$, the throughput of $A_k$ must be twice the throughput of $A_{k'}$. Our objective is then to maximize $\min_k \left\{ \frac{\rho^k}{\pi_k} \right\}$.

3.2.1. One-port model

We extend the linear program (LP) to several applications. For the one-port model, we get the following formulation:
We characterize each application \( A_k \) by its communication-to-computation ratio (CCR) \( \delta_k/w_k \): the larger the CCR, the more communication-intensive the application. This parameter has a critical influence on the shape of the solution. In an optimal solution, one can show that applications with larger CCR should be allocated to processors with larger bandwidth. Resources are split in ordered “slices”, each slice being devoted to the processing of a single application. Figure 3 illustrates the affinity property [5]: if applications are sorted in non-increasing order of CCR (\( \delta_1/w_1 \geq \delta_2/w_2 \geq \cdots \geq \delta_K/w_K \)) and processors are sorted in non-increasing bandwidth (\( b_1 \geq b_2 \geq \cdots \geq b_p \)), then there exist indices \( a_0, a_1, \ldots, a_K \) such that only processors \( P_u, u \in [a_k-1, a_k] \) execute tasks of application \( A_k \) in the optimal solution.

In [5] we have experimentally compared the following three simple algorithms:

- A pure demand-driven strategy, where the scheduler sends a task of any application to the first worker posting a request
- A coarse-grain strategy: we assemble all applications into a single big one and use the single-application bandwidth-centric algorithm explained in Section 3.1. For example, consider two applications with priorities \( \pi_1 = 3 \) and \( \pi_2 = 1 \). We gather the tasks into bundles where each bundle contains three tasks of application \( A_1 \) and one task of application \( A_2 \). We have now reduced the problem to scheduling a single, coarse-grain bag-of-tasks application.
- The affinity-based strategy, which relies on the above affinity property to pair application tasks and computation/communication resources.
The third strategy dramatically outperformed the first two. We expected the result for the first strategy. But it is insightful that the second strategy, although optimal for a single bag-of-tasks application, was not “clever” enough for several ones.

### 3.2.2. Bounded multi-port model

We now move to the multi-port model. As for the single-application case, we can easily adapt the linear program in order to cope with this model. Constraint (M-ii) is replaced by two constraints, one bounding the capacity of each edge:

\[ \forall i, \sum_{k=1}^{K} \alpha_i^k \frac{\delta_k}{b_i} \leq 1, \quad \text{(M-ii-a)} \]

and one bounding the network capacity of the master:

\[ \sum_{i=1}^{p} \sum_{k=1}^{K} \alpha_i^k \delta_k B \leq 1. \quad \text{(M-ii-b)} \]

Although the affinity between applications and processors does not result in a slicing property as in the one-port model, it still has a big impact on the optimal solution.

Consider the simple problem described in Figure 4(a), with two processors and two applications of same priority. Processor \( P_1 \) has a large computation speed but a small bandwidth, while it is the opposite for \( P_2 \). Application \( A_1 \) is computation-intensive with CCR 1/10, while \( A_2 \) is communication-intensive, with CCR 10. In the optimal solution given by the linear program, tasks of application \( A_2 \) are executed only by processor \( P_1 \), while processor \( P_2 \) is in charge of all tasks of application \( A_1 \). This results in a throughput of one task per time unit for each application.

If we gather both applications into a coarse-grain application, we get tasks composed of one task of \( A_1 \) and one task of \( A_2 \). The parameters of the coarse-grain application are \( \delta_{CG} = 11 \) and \( w_{CG} = 11 \). Each processor is only able to process 1/11 tasks during each

---

### Figure 4. Example of multiple applications with multi-port model.

<table>
<thead>
<tr>
<th>throughput</th>
<th>for ( A_1 ) on ( P_1 )</th>
<th>for ( A_2 ) on ( P_2 )</th>
<th>objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimal</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>coarse-grain</td>
<td>1/11</td>
<td>1/11</td>
<td>1/11</td>
</tr>
<tr>
<td></td>
<td>1/11</td>
<td>1/11</td>
<td>1/11</td>
</tr>
</tbody>
</table>

(b) Throughput achieved by both strategies.
time unit, and the throughput is dramatically decreased. Note that in this toy example, we have not bounded the network capacity of the master.

How would a pure demand-driven scheduler behave on this example? This is hard to predict, as it depends on the initial task sent to each processor: if, by chance, the scheduler sends a task of $A_2$ to $P_1$, and one of $A_1$ to $P_2$, then both tasks will be completed simultaneously and both processors will request some more work. If the scheduler repeats its initial choice, then it will reach the optimal throughput. On the contrary, if it makes the opposite choice (sending $A_1$ to $P_1$ and $A_2$ to $P_2$), then the processing of both tasks will be slowed down, leading to a throughput of $1/10$ for each of them.

Why take the risk? In a multi-application setting, demand-driven scheduling can be very unstable. It seems reasonable to approximate its average performance by considering the coarse-grain scheduler, which represents the case where the fraction of each application sent to a processor does not depend on the target processor. In the multi-port model, the demand-driven strategy gives the best throughput for a single application, so we could expect a good performance with the resulting coarse-grain application. However, we have shown that its performance can be significantly reduced because it does not consider the affinity between processors and applications.

For the one-port model, the demand-driven strategy performs poorly even with a single application. Due to the high complexity of the one-port model, one could argue that the problems come from the limited capacity of the master, and that it would be sufficient to enhance its network capacity, or even to duplicate the server in charge of sending tasks for the scheduling complexity to disappear. However, the example of Figure 4 shows that even with unlimited network resources on the master and with the simple multi-port model, executing multiple applications with a demand-driven strategy leads to sub-optimal performance.

3.3. Platform selection

We have shown the usefulness of a scheduler that performs resource selection, and assigns the best-suited application load to each enrolled resource: the throughput achieved for one or several bag-of-tasks applications is higher (and by an arbitrary factor) than that provided by demand-driven strategies. This observation holds true for the simplest possible platform, namely a single-level tree.

In practice the problem is more complicated. Either the platform is given, most likely in the form of a (hierarchical) multi-level tree, where each participating node has enrolled some neighboring resources. Or the platform is to be built out of, say, widely scattered and distributed resources (a cluster here, a supercomputer there, and a large network of workstations elsewhere). In the latter case, the user needs:

- either to extract the most efficient tree out of the general platform graph, which looks difficult because of the huge combinatorial set of possibilities to explore;
- or to deploy its application using the whole platform, which looks difficult too because the user will face in this case the complexity induced by cycles in the platform graph.

No need to go into technical details here, the reader will be easily convinced that every problem is indeed difficult. For a single bag-of-tasks application, the throughput achieved by the best tree can be arbitrarily bad compared to that of the general plat-
form [4]. With several bag-of-tasks applications on a fixed tree, complex algorithms must be used to achieve a good throughput [5].

4. Case study: matrix product under memory constraints

In this section we deal with matrix product, a key computational kernel in many scientific applications, which has been extensively studied on parallel architectures. Why revisit the mother-source of parallel algorithms? We address a grid-oriented client-server framework where the user enrolls participating resources (rather than using a fixed parallel machine). And just as we replaced makespan by throughput in the previous section, we replace makespan by communication volume in this case study so as to obtain a tractable problem. A thorough analysis allows us, first to find a good way to use the limited-memory, then to design a scheduling algorithm that is more efficient than demand-driven strategies.

4.1. Framework

Two well-known parallel versions are Cannon’s algorithm [15] and the ScALAPACK outer product algorithm [13]. Typically, parallel implementations work well on 2D processor arrays, because the input matrices are sliced horizontally and vertically into square blocks that are mapped one-to-one onto the physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor arrays. On a Grid, however, the computing resources are interconnected by a sparse network: there are no direct links between any pair of processors and assuming the interconnection network to be a 2D grid would lead to communication contentions and performance degradation. A new parallelization approach should thus be undertaken.

Furthermore, as the Grid may contain long-distance, and thus slow-communicating, network links, it becomes necessary to include the cost of both the initial distribution of the matrices to the processors and of collecting back the results. These input/output operations have always been neglected in the analysis of the conventional algorithms. This is because only $\Theta(n^2)$ coefficients need to be distributed in the beginning, and gathered at the end, as opposed to the $\Theta(n^3)$ computations$^1$ to be performed (where $n$ is the problem size). The assumption that these communications can be ignored could have made sense on dedicated parallel machines like the Intel Paragon, but it is no longer reasonable on heterogeneous platforms. Furthermore, when processors cannot store all the matrices in their memory, the total volume of communication required can be larger than $\Theta(n^2)$ as a same matrix element may have to be sent several times to a same processor.

We therefore adopt an application scenario where input files are read from a fixed repository (a disk on a data server). Computations will be delegated to available resources in the target architecture, and results will be returned to the repository. This calls for a master-worker paradigm, or more precisely for a computational scheme where the master

$^1$Of course, there are $\Theta(n^3)$ computations if we only consider algorithms that uses the standard way of multiplying matrices; this excludes Strassen’s and Winograd’s algorithms [17].
(the processor holding the input data) assigns computations to other resources, the workers. In this centralized approach, all matrix files originate from, and must be returned to, the master. The master distributes both data and computations to the workers. Finally, because we investigate the parallelization of large problems, we cannot assume that full matrix panels can be stored in worker memories and be re-used for subsequent updates (e.g., as in ScaLAPACK).

To summarize, the target platform is composed of several workers with different computing powers, different bandwidth links to/from the master, and different, limited, memory capacities. The first problem is resource selection. Which workers should be enrolled in the execution? All of them, or maybe only the faster computing ones, or else only the faster-communicating ones? Once participating resources have been selected, there remain several scheduling decisions to take: how to minimize the number of communications? in which order workers should receive input data and return results? what amount of communications can be overlapped with (independent) computations?

In Section 4.2, we state the scheduling problem precisely, and we introduce some notations. Next, in Section 4.3, we proceed with the analysis of the total communication volume that is needed in the presence of memory constraints. We show how to improve a well-known bound by Toledo et al. [32,23], and we outline an algorithm [28] almost achieving this bound on platforms with a single worker. We deal with homogeneous platforms in Section 4.4, and with heterogeneous ones in Section 4.5.

4.2. Framework and Notations

Here, we formally state the hypotheses on the application and on the target platform. We deal with the computational kernel \( C \leftarrow C + A \cdot B \). We partition the three matrices \( A, B, \) and \( C \) as illustrated in Figure 5. More precisely:

![Figure 5. Partition of the three matrices.](image-url)
We use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square blocks of size $q \times q$ (hence with $q^2$ coefficients). This is to harness the power of Level 3 BLAS routines [12]. Typically, $q = 80$ or $100$ when using ATLAS-generated routines [16,34].

The input matrix $A$ is of size $n_A \times n_{AB}$:

- we split $A$ into $r$ horizontal stripes $A_i$, $1 \leq i \leq r$, where $r = n_A/q$;
- we split each stripe $A_i$ into $t$ square $q \times q$ blocks $A_{i,k}$, $1 \leq k \leq t$, where $t = n_{AB}/q$.

The input matrix $B$ is of size $n_{AB} \times n_B$:

- we split $B$ into $s$ vertical stripes $B_j$, $1 \leq j \leq s$, where $s = n_B/q$;
- we split each stripe $B_j$ into $t$ square $q \times q$ blocks $B_{k,j}$, $1 \leq k \leq t$.

We compute $C = C + A \cdot B$. Matrix $C$ is accessed (both for input and output) by square $q \times q$ blocks $C_{i,j}$, $1 \leq i \leq r$, $1 \leq j \leq s$; there are $r \times s$ such blocks.

We point out that, with such a decomposition, all stripes and blocks have same size. This will greatly simplify the analysis of communication costs.

We target a star-shaped master-worker platform $S = \{M, P_1, P_2, \ldots, P_p\}$, composed of a master $M$ and of $p$ workers $P_i$, $1 \leq i \leq p$ (see Figure 1). We keep the notations of Section 3.1: worker $P_i$ has a computation speed $s_i$ and a communication bandwidth $b_i$. Because we manipulate large data blocks, we enforce a linear cost model as in Section 3.1, both for computations and communications (i.e., we neglect start-up overheads). In other words:

- It takes $X/s_i$ time-units to execute a task of size $X$ on $P_i$;
- It takes $X/b_i$ time units for the master to send a message of size $X$ to $P_i$ or to receive a message of size $X$ from $P_i$.

The target star platform is thus fully heterogeneous, both in terms of computations and of communications. Without loss of generality, we assume that the master has no processing capability (otherwise, add a fictitious extra worker paying no communication cost to simulate computation at the master). For the communication model, we once again use the one-port model. In summary:

- The master can only send data to, and receive data from, a single worker at a given time-step, and it cannot be enrolled in more than one communication at any time-step;
- A given worker cannot start an execution before it has terminated to receive the needed data from the master; similarly, it cannot start sending the results back to the master before finishing the computation.

Our final assumption is related to memory capacity; we assume that a worker $P_i$ can only store $m_i$ blocks (either from $A$, $B$, and/or $C$). For large problems, this memory limitation will considerably impact the design of the algorithms, as data re-use will be greatly dependent on the amount of available buffers.

### 4.3. Minimization of the communication volume

The classical objective is makespan minimization, that is the minimization of the overall execution time. Minimizing makespan, however, is very hard in our context. The only known results are for the degenerate case where $t = 1$ and there is a single worker with infinite memory. Even the problem complexity in the two-workers case is open [28]. Instead of targeting makespan minimization, we make the initial assumption that com-


communication costs dominate the problem. Therefore, we do not directly target makespan minimization, but communication volume minimization. Note that practical experiments reported in [28,27] do show that such a communication-volume minimization approach effectively leads to algorithms with shorter makespans.

We thus want to derive a lower bound on the total number of communications (sent from, or received by, the master) that are needed to execute any matrix multiplication algorithm. As we are only interested in minimizing the total communication volume, we can simulate any parallel algorithm on a single worker and, thus, we only need to consider the one-worker case.

We deal with the following formulation of the problem:

- The master sends blocks $A_{ik}$, $B_{kj}$, and $C_{ij}$,
- The master retrieves final values of blocks $C_{ij}$, and
- We enforce limited memory on the worker; only $m$ buffers are available, which means that at most $m$ blocks of $A$, $B$, and/or $C$ can simultaneously be stored on the worker.

First, we improve a lower bound on the communication volume established by Toledo et al. [32,23]. Then, we describe an algorithm that aims at re-using $C$ blocks as much as possible after they have been loaded, and we assess its performance.

### 4.3.1. Lower bound on the communication volume

To derive the lower bound, the idea is to estimate the number of computations made thanks to $m$ consecutive communication steps (once again, the unit here is a matrix block). Using Loomis-Whitney inequality [23], one can then show that a lower bound for the communication-to-computation ratio is:

$$\text{CCR}_{\text{opt}} \geq \sqrt{\frac{27}{8m}}.$$

### 4.3.2. The maximum re-use algorithm

The above lower-bound on the communication volume is obtained when the three matrices $A$, $B$, and $C$ are equally accessed during a sequence of communications. This may suggest to allocate one third of the memory to each of these matrices. In fact, Toledo [32] uses this memory layout. A closer look to the problem shows that the multiplied matrices, $A$ and $B$, have the same behavior which differs from the behavior of the result matrix $C$. Indeed, if an element of $C$ is no longer used, it cannot be simply discarded from the memory as the elements of $A$ and $B$ are, but it must be sent back to the master. Intuitively, sending an element of $C$ to a worker also costs the communication needed to retrieve it from the worker, and is thus twice as expensive as sending an element of $A$ or $B$. Hence the motivation to design an algorithm which reuses as much as possible the elements of $C$.

Cannon’s algorithm [15] and the ScaLAPACK outer product algorithm [13] both distribute square blocks of $C$ to the processors. Intuitively, squares are better than elongated rectangles because their perimeter (which is proportional to the communication volume) is smaller for the same area. We use the same approach here, even if there are no optimality results to justify it.

The maximum re-use algorithm uses the memory layout illustrated in Figure 6. Four consecutive execution steps are shown in Figure 7. Assume that there are $m$ available
buffers. First we find $\mu$ as the largest integer such that $1 + \mu + \mu^2 \leq m$. The idea is to use one buffer to store $A$ blocks, $\mu$ buffers to store $B$ blocks, and $\mu^2$ buffers to store $C$ blocks. In the outer loop of the algorithm, a $\mu \times \mu$ square of $C$ blocks is loaded. Once these $\mu^2$ blocks have been loaded, they are repeatedly updated in the inner loop of the algorithm until their final value is computed. Then the blocks are returned to the master, and $\mu^2$ new $C$ blocks are sent by the master and stored by the worker. As illustrated in Figure 6, we need $\mu$ buffers to store a row of $B$ blocks, but only one buffer for $A$ blocks: $A$ blocks are sent in sequence, each of them is used in combination with a row of $\mu$ $B$ blocks to update the corresponding row of $C$ blocks. This leads to the following sketch of the algorithm:

**Outer loop:** while there remain $C$ blocks to be computed:
- Store $\mu^2$ blocks of $C$ in worker’s memory: $\{C_{i,j} \mid i_0 \leq i < i_0 + \mu, j_0 \leq j < j_0 + \mu\}$.
- **Inner loop:** For each $k$ from 1 to $t$:
  1. Send a row of $\mu$ elements $\{B_{k,j} \mid j_0 \leq j < j_0 + \mu\}$;
  2. Sequentially send $\mu$ elements of column $\{A_{i,k} \mid i_0 \leq i < i_0 + \mu\}$. For each $A_{i,k}$, update $\mu$ elements of $C$.
- Return results to master.

The performance of one iteration of the outer loop of the *maximum re-use* algorithm can readily be determined:
- We need $2\mu^2$ communications to send and retrieve $C$ blocks.
- For each value of $t$:
  * we need $\mu$ elements of $A$ and $\mu$ elements of $B$;
  * we update $\mu^2$ blocks.

In terms of block operations, the communication-to-computation ratio achieved by the algorithm is thus

$$\text{CCR} = \frac{2\mu^2 + 2\mu t}{\mu^2 t} = \frac{2}{t} + \frac{2}{\mu}.$$
For large problems, i.e., large values of $t$, we see that CCR is asymptotically close to the value $CCR_\infty = \frac{2}{\sqrt{m}}$. We point out that, in terms of data elements, the communication-to-computation ratio is divided by a factor $q$. Indeed, a block consists of $q^2$ coefficients but an update requires $q^3$ floating-point operations. Also, the ratio $CCR_\infty$ achieved by the maximum re-use algorithm is lower by a factor $\sqrt{3}$ than the ratio achieved by the blocked matrix-multiply algorithm of [32]. Finally, we remark that the performance of the maximum re-use algorithm is quite close to the lower bound derived earlier:

$$CCR_\infty = \frac{2}{\sqrt{m}} = \sqrt{\frac{32}{8m}}.$$  

### 4.4. Algorithms for homogeneous platforms

We now adapt the maximum re-use algorithm to fully homogeneous platforms. We have a limitation on the memory capacity. So we must first decide which part of the memory will be used to store which part of the original matrices, in order to maximize the total number of computations per time unit.

We load into the memory of each worker $\mu$ blocks of $A$ and $\mu$ blocks of $B$ to compute $\mu^2$ blocks of $C$ (in other words, we waste some memory in order to decrease the number of communications and the synchronization effects). In addition, we need $2\mu$ extra buffers, split into $\mu$ buffers for $A$ and $\mu$ for $B$, in order to overlap computation and communication steps. In fact, $\mu$ buffers for $A$ and $\mu$ for $B$ would suffice for each update, but we need to prepare for the next update while computing. Overall, the number $\mu^2$ of $C$ blocks that we can simultaneously load into memory is defined by the largest integer $\mu$ such that $\mu^2 + 4\mu \leq m$.

We have to determine the number of participating workers $\Psi$. On the communication side, we know that in a round (computing a $C$ block entirely), the master exchanges with each worker $2\mu^2$ blocks of $C$ ($\mu^2$ sent and $\mu^2$ received), and sends $\mu t$ blocks of $A$ and $\mu t$ blocks of $B$. Also during this round, on the computation side, each worker computes $\mu^2 t$ block updates. If we enroll too many processors, the communication capacity of the master will be exceeded. There is a limit on the number of blocks sent per time unit, hence on the maximal processor number $\Psi$, which we compute as follows: $\Psi$ is the smallest integer such that the total communication time from the master to all workers exceeds the computation time of each worker. We derive

$$2\mu tc \times \Psi \geq \mu^2 tw$$  

where $c = q^2/b$ and $w = q^3/s$ respectively represent the communication and computation times for a $q \times q$ matrix block. Finally, we cannot use more processors than are available, hence we obtain the formula

$$\Psi = \min \left\{ \frac{\mu q b}{2s}, p \right\}$$  

Finally, the participating workers receive data in a round-robin fashion.

\textsuperscript{2}For simplicity, we group $\mu$ messages of size 1 into one single message of size $\mu$, at the price of a small increase in memory requirement.
4.5. Algorithms for heterogeneous platforms

We now consider the general problem, i.e., when processors are heterogeneous in terms of memory size as well as computation and/or communication time. As in the previous section, $m_i$ is the number of $q \times q$ blocks that fit in the memory of worker $P_i$, and we need to load into the memory of $P_i$ $2\mu_i$ blocks of $A$, $2\mu_i$ blocks of $B$, and $\mu_i^2$ blocks of $C$. This number of blocks loaded into the memory changes from worker to worker, because it depends upon their memory capacities. We first compute all the different values of $\mu_i$, $\mu_i$ being the largest integer such that: $\mu_i^2 + 4\mu_i \leq m_i$. To adapt the maximum re-use algorithm to heterogeneous platforms, the first idea would be to adopt a steady-state-like approach. The problem with such a solution, however, is that workers may not have enough memory to execute it! Therefore, this solution cannot always be realized in practice and, to avoid such memory problems, resource selection will be performed through a step-by-step simulation. However, we point out that a steady-state solution can be seen as an upper bound of the performance that can be achieved.

The different memory capacities of the workers imply that we assign them chunks of different sizes. This requirement complicates the global partitioning of the $C$ matrix among the workers. To take this into account, while simplifying the implementation, the algorithm only assigns full matrix column blocks. This is done in a two-phase approach.

In the first phase allocation of blocks to processors is pre-computed, using a processor selection algorithm later described. We start as if we had a huge matrix of size $\infty \times \sum_{i=1}^{p} \mu_i$. Each time a processor $P_i$ is chosen by the processor selection algorithm it is assigned a square chunk of $\mu_i^2 C$ blocks. As soon as some processor $P_i$ has enough blocks to fill up $\mu_i$ block columns of the initial matrix, we decide that $P_i$ will indeed execute these columns during the parallel execution. Therefore we maintain a panel of $\sum_{i=1}^{p} \mu_i$ block columns and fill them out by assigning blocks to processors. We stop this phase as soon as all the $r \times s$ blocks of the initial matrix have been allocated columnwise by this process. Note that worker $P_i$ will be assigned a block column after it has been selected $\lceil \frac{r}{\mu_i^2} \rceil$ times by the algorithm.

In the second phase we perform the actual execution. Messages will be sent to workers according to the previous selection process. The first time a processor $P_i$ is selected, it receives a square chunk of $\mu_i^2 C$ blocks, which initializes its repeated pattern of operation: the following $t$ times, $P_i$ receives $\mu_i A$ and $\mu_i B$ blocks, which requires $2\mu_i c_i$ time-units.

To decide which processor to select at each step of the first phase, one can imagine two variants of an incremental algorithm, a global one that aims at optimizing the overall communication-to-computation ratio, and a local one that selects the best processor for the next stage.

**Global selection algorithm.** The intuitive idea for this algorithm is to select the processor that maximizes the ratio of the total work achieved so far (in terms of block updates) over the completion time of the last communication. The latter represents the time spent by the master so far, either sending data to workers or staying idle waiting for the workers to finish their current computations. Estimating computations is easy: $P_i$ executes $\mu_i^2$ block updates per assignment. Communications are slightly more complicated to deal with; we cannot just use the communication time $2\mu_i c_i$ of $P_i$ for the $A$ and $B$ blocks because we need to take its ready time into account (here $c_i = q^2/b_i$ is the communica-
tion time for $P_i$ to receive one block). Indeed, if $P_i$ is currently busy executing work, it cannot receive additional data too much in advance because its memory is limited.

**Local selection algorithm.** The global selection algorithm picks, as the next processor, the one that maximizes the ratio of the total amount of work assigned over the time needed to send all the required data. Instead, the local selection algorithm chooses, as destination of the $i$-th communication, the processor that maximizes the ratio of the amount of work assigned by this communication over the time during which the communication link is used to performed this communication (i.e., the elapsed time between the end of $(i - 1)$-th communication and the end of the $i$-th communication). As previously, if processor $P_j$ is the target of the $i$-th communication, the $i$-th communication is the sending of $\mu_j$ blocks of $A$ and $\mu_j$ blocks of $B$ to processor $P_j$, which enables it to perform $\mu_j^2$ updates.

The description of the global and local selection algorithms is sketchy here, please see [27] for further details.

**4.6. Conclusion**

In this section, our aim was to give an insight on the difficulty of the problem with heterogeneous resources: selecting which ones to enroll, and partitioning the matrices into patterns of different sizes and shapes, turns out to be unexpectedly challenging. However, this theoretical study paid-off. Indeed, through MPI experiments, we have been able to show [27] that our algorithm for heterogeneous platforms has far better performance than solutions using the memory layout proposed in [32]. Furthermore, our static heterogeneous algorithm has slightly better average performance than dynamic algorithms using the same memory layout, but uses fewer processors, and has a far better worst case. Overall, it is far more efficient.

**5. Multi-criteria scheduling**

So far we focused on mono-objective problems: maximizing the throughput, i.e., the number of tasks processed per time unit (Section 3) or minimizing communication volume (Section 4).

In practice, and even for simple BOINC-like applications, several other important optimization criteria should be considered to fulfill users expectations. In the following, we start by introducing workflow applications, which consist of pipelined DAGs (rather than independent tasks). Then we list many possible objectives for these applications, and we discuss how schedulers can deal with several (usually conflicting) objectives simultaneously.

**5.1. Structured workflows**

A bag-of-tasks application (Section 3) is a collection of identical independent tasks. A workflow is a collection of identical task graphs, or DAGs (hence a single bag-of-tasks application is a workflow whose DAG reduces to a single node). Workflows naturally arise in many frameworks. Take the example of a JPEG encoder. You can-
not apply the Fast Discrete Cosine Transformation \cite{33} on your JPEG encoder (see http://www.jpeg.org/) before some pre-processing on the image: scaling, color space conversion, and so on. As a consequence, the application graph of the JPEG encoder is a linear chain of processings to be applied successively on each incoming image. Of course, rather than just chains, we can have fork or fork-join graphs, or series-parallel graphs, or even arbitrary DAGs.

Classical scheduling aims at minimizing the makespan of a single DAG: a single data set goes through the application graph. With workflows we have pipelined DAGs, because we operate on a collection of data sets that are processed in a pipeline fashion. Each data set is an input to the application graph and traverses it until its processing is complete. Several data sets can be processed concurrently. Mapping and/or scheduling consists in assigning tasks to resources, so as to minimize one or several objectives. Again, a task (also called a stage) is in fact the same processing that must be applied to all the elements in the input data set. Hence, a task corresponds to a collection of numerous identical computations, each applied to a different data set (think of the images entering the JPEG encoder).

5.2. Objective functions

For workflow applications, the first objective that comes to mind is throughput maximization: the goal is to process as many data sets per time unit as possible. However, looking back at classical scheduling, makespan minimization was an important objective too. This remains true for workflows, and in particular for real-time applications. The definition must be adapted, and we talk of latency rather than of makespan, in order to avoid confusion. The latency is the time elapsed between the beginning and the end of the execution of a given data set, hence it measures the response time of the system to process the data set entirely. Note that it may well be the case that different data sets have different latencies (because they are mapped onto different processor sets), hence the latency is defined as the maximum response time over all data sets. Note also that minimizing the latency is antagonistic to maximizing the throughput. For a linear chain application, latency is minimized by assigning the whole application to a single processor, thus working in a fully sequential way: no communication is paid. However throughput can be increased by distributing tasks over processors and working in a pipelined manner. Already we guess that trade-offs will have to be found between these criteria. Indeed, several work dealt with both these criteria, for instance see \cite{31,7}.

With the advent of large-scale heterogeneous platforms, resources may be cheap and abundant, but resource failures (processors/links) are more likely to occur and have an adverse effect on the applications. Not only every user is quite likely to face unrecoverable hardware failures when deploying applications on clusters or grids \cite{20,21,1,18}, but unrecoverable interruptions can also take place in other important frameworks, such as loaned/rented computers being suddenly reclaimed by their owners, as during an episode of cycle-stealing \cite{2,11,29}. Consequently, there is an increasing need for developing reliable schedules. Another optimization criterion that could be maximized is then the reliability of the schedule, given a failure model for the resources.

Another trendy objective emerges for current platforms, namely the energy minimization objective. Green scheduling aims at minimizing energy consumption, by running processors at lower frequencies \cite{3}, or by reducing the number of processors en-
rolled. Of course being green often involves running at a slower pace, thereby reducing the application throughput.

In addition to being green, one may also want to reduce the execution cost. The cost may be simply expressed in terms of the number of resources enrolled to deploy the application; or it can further be refined if the user would pay pay for processing units, memory cards, network cards, and so on. We may also have a rental cost that depends upon both resources and rental duration. In all cases, the execution cost of the platform another objective that can be considered, which is antagonist to performance related objectives (with fewer processors, you are less fast and/or reliable).

Finally, even more objectives will appear in a multi-application setting. It was easy with several bag-of-tasks applications, because we assumed fixed priority factors (Section 3.2). More generally, some form of fairness must be guaranteed between all the applications. Typical measures are the maximum stretch of an application or the sum of all application stretches [6]. The stretch of an application is the slowdown factor incurred by its execution time when sharing resources with the other applications. Add different release dates and deadlines for each application, and contemplate the difficulty of this scheduling problem!

5.3. Dealing with multi-criteria

How to deal with so many objective functions? In traditional approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But is it natural for the user to maximize the quantity $0.7T + 0.3R$, where $T$ is the throughput and $R$ the reliability? What about adding latency and energy parameters into the story? Obviously, the problem here is that we mix apples and bananas: the criteria are very different in nature and it does not make much sense for a user to make a linear combination of them.

Users are more likely to ask questions like "I want a frame rate $T$ and a response time $L$ for my JPEG encoder, what is the least amount of energy that I will consume?". Thus we advocate the use of multi-criteria with thresholds. To give another example, we would aim at maximizing the throughput of the application, but accepting only schedules whose reliability is at least 99%. Now, each criteria combination can be handled in a natural and meaningful way: one single criterion is optimized, under the condition that a threshold is enforced for all other criteria.

Several interesting trade-offs appear when dealing with multi-criteria optimizations. Let us illustrate one of them with a little case study: the application graph is a linear chain, and we target throughput and reliability objectives. In order to increase reliability, a solution consists in replicating a task, or set of tasks, onto several resources. Then each data set is entirely processed by several resources, and if some of the resources fail during execution, the processing is not interrupted. In the extreme case, we could replicate the whole chain onto each resource, and even if all processors but one fail, we still get the result. However, the throughput would be very low for such a highly reliable schedule. For throughput maximization, we would rather split the chain and assign each task to a different processor, in order to process different data sets in parallel. Moreover, we can also replicate each task onto several processors, but this time to increase the throughput: for instance if we replicate a task on two processors, the first one would process even-numbered data sets, while the second processor would process odd-numbered data sets.
In the most favorable case, doing so would double the throughput. Of course, such a solution is much less reliable because a single failure stops the whole application.

In such situations, some knowledge of the application and platform parameters may help the scheduler decide which tasks to group onto the same processor set ands for each processor set, which processors are doing replication for reliability and which ones are doing replication for throughput. Needless to say, the story becomes even more complex when adding more objectives, and when tackling applications whose graph is an arbitrary DAG. Demand-driven strategies are quite likely to fail, even with an infinity of resources.

6. Conclusion

In this chapter, we have explained that scheduling remains a mandatory activity, despite the advent of cheaper and ubiquitous resources.

We started with a glance at BOINC-like applications, introducing steady-state scheduling. Through this case study, we advocated the importance of going divisible (using non-integer number of tasks). Schedules can then be expressed in a compact periodic manner, as opposed to the full-length schedule descriptions of classical scheduling. Despite the simplicity of the problem, we have shown the importance of resource selection, even if resources are abundant and cheap.

Then we revisited the matrix product kernel under memory constraints. Instead of makespan minimization (too complicated) we focused on communication volume and derived efficient algorithms that either minimize resource usage (homogeneous platforms) or squeeze the most out of collections of processors with different memory capacity (heterogeneous platforms).

Finally, the story got more complicated with the introduction of multi-criteria scheduling: makespan minimization is not relevant enough in most situations. Users also care about throughput, reliability, energy, fairness, and so on. However, rather than linear combinations, it makes much more sense to optimize only one criterion, given that a threshold is enforced for the others. Often the criteria are antagonistic, which leads to many algorithmic challenges to tackle.

Altogether we gave several examples for which the design of a good scheduling algorithm was a “sine-qua-non” to obtain good performance. Of course, problems are even more complicated in real life, and the scheduler gets even more useful. Further techniques can be developed if the knowledge of the platform and/or of the application is only partial or not fully accurate. One can schedule pipelined applications by phases and re-inject currently acquired knowledge to drive the scheduling decisions for future phases, thereby exploiting up-to-date parameter information. Otherwise, if the platform parameters are subject to variations (not to speak of unrecoverable interruptions), we can design robust algorithms able to react to these variations, through the use of stochastic models.

With the advent of multicores, and more importantly of clusters of multicores, additional problems will arise. Schedulers will have to cope with new locality rules, and will have to trade-off between (fast but scarce) memory accesses and (slower but unlimited) network communications. Most likely, yet another level of hierarchy (outermost tiling) will be needed. We intend to address these forthcoming algorithmic challenges. As claimed in the introduction, we do view a bright future for schedulers!
Acknowledgement. Special thanks go to Arnaud Legrand for several enlightening discussions. Several ideas exposed here matured through joint work with Olivier Beaumont, Larry Carter, Henri Casanova, Jack Dongarra, Jeanne Ferrante, and Arnold Rosenberg. We also acknowledge the contributions of our PhD students Matthieu Gallet, Jean-François Pineau, and Veronika Rehn-Sonigo. We express our gratitude to all of them.

References


This page intentionally left blank
Chapter 2
Architectures
High Performance Computing with FPGAs

Erik H. D’HOLLANDER a,1, Kristof BEYLS b

a Ghent University
Department of Electronics and Information Systems
9000 Ghent, Belgium
b ARM Ltd., Cambridge, UK

Abstract
Field-programmable gate arrays represent an army of logical units which can be organized in a highly parallel or pipelined fashion to implement an algorithm in hardware. The flexibility of this new medium creates new challenges to find the right processing paradigm which takes into account the natural constraints of FPGAs: clock frequency, memory footprint and communication bandwidth. In this paper first the use of FPGAs as a multiprocessor on a chip or its use as a highly functional coprocessor are compared, and the programming tools for hardware/software codesign are discussed. Next a number of techniques are presented to maximize the parallelism and optimize the data locality in nested loops. This includes unimodular transformations, data locality improving loop transformations and use of smart buffers. Finally, the use of these techniques on a number of examples is demonstrated. The results in the paper and in the literature show that, with the proper programming tool set, FPGAs can speed up computation kernels significantly with respect to traditional processors.

Keywords. FPGA, data locality, high performance, loop transformations

Introduction

Modern FPGAs consist of logic blocks, gates, memories, ALUs and even embedded processors, which can be arbitrarily interconnected to implement a hardware algorithm. Because of their flexibility and growing capabilities, field programmable gate arrays are the topic of intensive research in high-performance computing.

The high reconfigurability and inherent parallelism creates a huge potential to adapt the device to a particular computational task. At the same time, the embodiment of a hardware algorithm constitutes a departure from the classical Von Neumann or Harvard processor architecture. New computing paradigms need to be explored in order to exploit FPGAs to their full potential [8,22]. This involves a thorough knowledge of the design constraints, the development environment, the hardware description tools and the methodology for mapping an algorithm onto the hardware. On the other hand, traditional performance metrics such as instructions per cycle, clock speed, instructions

1Corresponding author: Erik H. D’Hollander, e-mail: dhollander@elis.ugent.be
per second, have no direct meaning for an algorithm that is directly executed in hardware [26,21,35]. In order to give an overview of the possibilities and challenges of this new and rapidly evolving technology, the following topics will be covered. First the hardware of the FPGAs is discussed. Next the different constellations of FPGAs used in high-performance computing are explored. An important aspect is the software tools required to configure the hardware, in particular the number of high-level languages which facilitate hardware-software codesign. Finally, a number of program transformations are described which improve the execution speed, data locality and performance of the hardware designs.

1. Architecture, computing power and programmability

Semantically, a Field-Programmable Gate Array (FPGA) means an array of gates which can be arbitrarily interconnected by a configuration program that describes the paths between the logic components. After programming, the FPGA is able to implement any logic function or finite state machine (see fig. 1).

![Figure 1. Schematic representation of an FPGA architecture: CLBs (configurable logic blocks) and interconnection network.](image)

Since its conception in the 1980s, the granularity and the complexity of the logic blocks has much evolved. Apart from the fine-grained AND, XOR and NOT gates, the most important blocks are lookup tables or LUTs, which are able to implement a logic function of 4 to 6 inputs. Flip-flops are able to store the state of a calculation, and turn the FPGA into a finite state machine. In addition to the fine-grained cells, most FPGAs have medium grain four-bit ALUs which can be joined to form ALUs of arbitrary precision, as well as coarse grain components, e.g. word size ALUs, registers and small processors with an instruction memory. All these components can be arbitrarily interconnected using a two-dimensional routing framework. More recently, heterogeneous FPGAs contain embedded multiplier blocks, larger memories and even full-fledged microprocessors, such as the PowerPC in the Vertex family of Xilinx, all integrated in one chip. These developments make the FPGA all the more attractive.
1.1. FPGAs versus ASICs

The ‘algorithm in hardware’ concept may also be implemented in an application-specific integrated circuit. ASICs are used to improve the computational tasks by implementing algorithmic functionality into optimized HPC hardware, e.g. a JPEG encoder [28].

Similar design steps

Both FPGAs and ASICs have a similar design cycle, which involves the behavioral description of the hardware, the synthesis, placement and routing steps. First the algorithm is described in a hardware description language, such as VHDL2 or Verilog. Next the behavioral description is converted into a hardware realization using logic synthesis. This step is comparable to the translation of a high-level program into machine instructions. However the results of the translation process is not an executable program, but an algorithm implemented in hardware. The synthesis step uses the logic components available in the FPGA or the ASIC target, similar to a compiler using the machine instructions available in a processor. Next, the components are mapped or placed onto the substrate and finally in the routing step the components are interconnected.

Different design implementation

There are significant differences between FPGAs and ASICs.

- The major advantage of an FPGA is that the design cycle is greatly reduced. The synthesis, mapping and routing is done using an integrated toolset which produces a bitstream file. The bitstream file describes the way in which the logic components are interconnected and configured. This file is loaded into static RAM and programs the FPGA at start up in a number of milliseconds. The whole process from hardware description to bitstream generation takes an order of minutes or hours. On the other hand, the development of an ASIC prototype requires weeks or months for a comparable design cycle. The prototype has to be tested and validated, and possibly resubmitted to generate a new design.
- The FPGA is reconfigurable, implementing a new bitstream means a new hardware design. Therefore, FPGAs can be reused for a new task in an HPC system. The reconfiguration of an FPGA can even occur at runtime. Using reprogramming, the same FPGA can be used in different phases of a program to carry out compute intensive tasks.
- On the other hand, ASICs consume less power, less area, and deliver a higher performance than FPGAs. This is not a critical limitation, since successful static or fixed FPGA designs may be burned into ASICs.

1.2. FPGA characteristics and constraints

The versatility of FPGAs comes at a price: there is no clear-cut computing paradigm for FPGAs. Reconfigurable computing may embody the traditional processor as a soft-core processor, act as an independent finite state machine, or operate as a special-purpose processor in conjunction with a CPU or even other FPGAs.

---

2VHDL = VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
FPGAs typically have a small onboard memory, no cache and a slow connection to external memory. Although onboard memory is a critical resource (typically less than 100Mb block RAM or BRAM), the memory is dual-ported and the aggregated memory bandwidth is in order of 100 GB/s [39]. The absence of a cache avoids cache coherency problems, but the nonuniform memory hierarchy has to be taken into account. The use of a high-speed scratchpad memory may be used to transparently optimize the data access [3,25]. Novel techniques such as data reuse analysis in conjunction with a scratchpad memory [23] have shown to outperform cache based architectures. FPGAs are bound to optimize data locality for an efficient operation, e.g. by reusing data in fast memory, and optimizing the use of registers, see section 7.

Most FPGA designs are limited to fixed point arithmetic, because floating-point designs are complex. Furthermore, the clock frequency depends on the characteristics of the design and is typically one order of magnitude lower than the clock frequency of current processors.

To summarize, the versatility and programmability, together with the unbounded parallelism make FPGAs an ideal choice to map parallel compute kernels onto fast dedicated hardware designs. Topical issues are:

- what computing paradigm is most suitable?
- which approach is best fitted to program HPC FPGAs?
- how to partition a program into a hardware (FPGA) and software (processor) codesign?
- what type of applications benefits most from a hardware/software architecture?
- which program transformations optimize FPGA-based parallelism and data locality?

These questions are addressed in the following sections, and answers, results and new concepts from academia and industry are presented.

2. FPGA computing paradigms

2.1. FPGA as multiprocessor: MPSoC

Traditional computer architectures are built using a data path and a control path. In a typical instruction cycle, the control decodes the instruction and configures the data path; the data path performs the operations on the data and actually executes the instruction. The configuration of the data path by the control ensures that the proper instruction is executed. The flexibility and size of the FPGAs allows to duplicate the processor architecture as a soft core circuit. Modern FPGAs allow to instantiate 20 and more soft core processors, or soft multiprocessors [24]. Attractive as they may be, these designs face competition with hardware Multiprocessor Systems on a Chip (MPSoCs), which have a higher performance and lower energy consumption. As a matter of fact, the microprocessor companies deliver multicores today and announced that multicore performance is building up [10]. However, the reconfigurability of FPGAs make this platform an attractive testbed for future multicores [31]. The RAMP Gold (Research Accelerator for Multiple Processors) uses a multithreaded Sparc instruction set to run applications on a multicore prototype architecture simulated on an array of Xilinx Virtex FPGAs.
Message passing or shared memory?

Multiple processors communicate and collaborate either by storing data and semaphores in shared memory or by exchanging messages over dedicated channels. A shared memory between two or more processors requires that there is one common address space. Setting up a shared memory involves first an addressing logic which accesses the block RAMs spread over the FPGA, and secondly a common bus with arbitration logic controlling the access of the different processors to the shared memory. In [38] a shared memory based multiprocessor with an hierarchical bus structure has been presented, yielding a speedup of 3.2 on 4 processors. In [2] a testbed is developed to emulate a multiprocessor with shared memory and a cache coherency protocol. The system is used as a research environment to study new parallel architectures.

Despite the rising interest in multi-cores and multiprocessors on a chip, a multi-processor on an FPGA is bound to face difficulties due to lack of support for a large shared memory. In contrast, the regularity of the logic and interconnections on an FPGA makes it more suitable to function as a pipelined data path instead of implementing a full-functional processor. In fact, most FPGA developers have opted to provide FPGAs with embedded processors, including intellectual property (IP) cores for several types of buses linking the embedded processor with the rest of the FPGA. Examples are the On-chip Peripheral Bus (OPB) and Fast Simplex Link (FSL) buses for the PowerPC processor in the Virtex Xilinx FPGAs. However, no arbitration logic is yet provided for a multi-master bus, and most solutions opt for a point-to-point link between the processors, using message passing.

Considering the message passing alternative, FPGAs have many input-output connections and are very suitable to operate as processing elements in a network of computers, or as an accelerator for dedicated operations in a supercomputer, such as the Cray XD1 [36]. In this area several initiatives have been taken to implement the message passing interface, MPI, as a communication fabric onto the FPGA. This allows FPGAs to communicate with each other using well-defined standards. Examples are TMD-MPI [32] implementing a lightweight subset of MPI, which is designed for systems with limited or no shared memory, no operating system and low processing overhead, such as Toronto’s TMD scalable FPGA-based multiprocessor [29]. A multi-FPGA system consists of 9 soft MicroBlaze processors interconnected with a Fast Serial bidirectional Link (FSL) on a single FPGA and a megabit transceiver (MGT) connecting several FPGAs. A speedup of 32 on 40 processors executing Jacobi’s algorithm for the heat equation is reported [33]. In [40] an FPGA based coprocessor is presented that implements MPI primitives for remote memory access between processors of a multiprocessor.

2.2. FPGA as accelerator

Since on-chip memory is limited, large data sets need to be stored off-chip, e.g. in a processor’s memory. Furthermore, using the hardware/software codesign concept, an FPGA can be easily configured to perform specialized computations using a tailored hardware algorithm. One of the major bottlenecks in such a configuration is the communication bandwidth gap between the processor running at GHz and the FPGA running at MHz speed. Cray uses the RapidArray communication processor which delivers a 3.2 GB/s bidirectional interconnect with 6 Virtex-II Pro FGPAs per chassis [9]. This allows to
download a computation kernel, such as a tiled matrix multiply, which executes 9 times faster on the FPGAs than on the XD1’s own Opteron CPU [6].

Silicon Graphics uses the RC100 blade [34] to speed up applications with reconfigurable computing. A blade contains dual Xilinx Virtex 4 LX200 FPGAs and the bandwidth to the processor memory is 6.4 GB/s.

Another area where the FPGA excels is its use as simulator of new computer architectures. The design of new architectural features such as prediction buffers, cache sizes, number and function of processing elements requires a cycle-accurate simulation. By construction, FPGAs are excellently placed to emulate many of the features studied in new architectural designs. The use of FPGAs to speed up the simulations has two major advantages:

1. the emulation is orders of magnitude faster than simulation;
2. the emulated architecture can be monitored directly with hardware probes located in the FPGA at the points of interest.

The whole concept of using FPGAs for hardware design is present in the RAMP prototyping system [2], which serves as a testbed for a large number of projects. In another publication [37], the simulation of multiprocessors or CMPs is expedited by orders of magnitude using FPGAs.

2.3. **FPGA as high-performance coprocessor**

A proficient use of FPGAs is as a collaborative specialized hardware algorithm running under control of a separate processor. This is the best of both worlds: a processor is used to control the flow of the algorithm and the FPGA is used to execute a specialized fast hardware implementation of the algorithm. High performance computers are equipped with reconfigurable FPGAs to operate as fast specialized computing elements, accessed, controlled and addressed by the associated processor. The FPGAs are either connected to a global shared memory (e.g. the SGI-Altix/RASC [34]) or directly to a companion high performance processor (e.g. the Cray-XD1 [9]). The first configuration has more possibilities to assign a farm of FPGAs for a particular job, while the second organization provides a faster and direct link between the processor and the associated FPGA. Both approaches have proven to be able to provide several orders of magnitude speedup on the right applications, e.g. DES breaking or DNA and protein sequencing [16].

To obtain these huge speedups requires a blend of a number of hardware, software and application characteristics, which are summarized as follows.

- A tight high-speed, low latency connection between processors and FPGAs.
- A compute-intensive problem with inherently massive regular parallelism in fixed-point arithmetic. Floating-point IP cores take a lot of chip area and are much slower than integer arithmetic.
- The problem must have good data locality. The data movement between the processor and the FPGA as well as within an FPGA remains slow with respect to the low-level parallel operations in the FPGA. Many successful FPGA applications operate in a single program multiple data (SPMD) fashion, by replicating a compute kernel many times and processing streams of data. Since FPGAs have a limited amount of memory, the data used for a computation should be fetched only once, such that all the computations on the data are finished in one computing step and then the data can be discarded.
There is a need for a high level programming environment which is able to seamlessly integrate the hardware/software codesign.

The problem lends itself easily to a hardware/software partitioning in which the software runs on the host and the hardware is executed on the FPGA.

3. FPGA programming languages

The lowest language level for programming FPGAs is VHDL. However VHDL is too hardware specific and can be regarded as the assembly language of reconfigurable computing. The lowest system-level language in wide use is C. In recent years many C-dialects have been developed to program FPGAs. Examples are Handel-C [1,27], Streams-C [17], Impulse-C [30], SystemC [18] and SPARK [20]. The major difference between these languages is their adherence to the ANSI C standard on the one hand or a C-like language with new pragmas and keywords for hardware synthesis on the other hand. The advantage of all approaches over VHDL or Verilog is the higher level of abstraction, which fosters more productivity.

The most well known C-like language with a strong hardware flavor is SystemC. SystemC is managed by the open SystemC Initiative (OSCI), and the language was recently recognized as IEEE standard Std 1666 2005. SystemC exists as a C++ class library which permits hardware-specific objects to be used within C++. The specific SystemC identifiers start with "sc_". In particular, events, sensitivity, modules, ports and hardware data types allow to express concurrency, signals and hardware structure. Handel-C has its roots in Occam, a language using the communicating sequential processes (CSP) paradigm to program transputers. Handel-C uses specific keywords to specify hardware, e.g. signals, interfaces, memory types and clocks. In [7] a test is discussed where a team of software developers without prior hardware knowledge develops an image processing algorithm in hardware using the Handel-C to VHDL compiler. The resulting hardware runs 370 times faster than the software algorithm.

Languages which comply with ANSI C mostly adhere to a well-defined computing paradigm such that the machine dependent characteristics can be mostly hidden for the programmer. An example of such a language is Streams-C or its descendant Impulse C, where the hardware part is programmed as a C function and compiled for execution on an FPGA. The communication between the software process and the hardware algorithm is based on streams of data. The code generation for the processor and the configuration generation for the FPGA is produced in one compile phase. Impulse C also generates the stubs to mediate the traffic between processor and FPGA. Impulse C is one of the languages available on the Cray and SGI supercomputers with FPGA accelerators.

4. Program transformations for parallelism

High-performance computing with FPGAs is most successful for applications with high parallelism and high data locality. In addition the compute kernel running on the FPGA should have a limited control flow in order to avoid pipeline stalling or synchronization overhead. In the area of bioinformatics or cryptography there are applications with obvious parallelism. However many common programs with inherent parallelism need to be
transformed in order to map the computations on an FPGA. In these cases nested loops are often a prime candidate for a hardware implementation when the proper conditions are met. Even when there is no obvious parallelism, there exist methods to reorient the computations in order to improve the parallelism and data locality at the same time. In the following, an approach using unimodular transformations is presented, which finds the maximum number of parallel iterations and orders the computations to optimize the data locality.

4.1. Loops with uniform dependencies

Consider a perfect loop nest $L$:

\[
\begin{align*}
&\text{for} \{ I_1 = 1..n_1 \} \\
&\text{for} \{ I_2 = 1..n_2 \} \\
&\ldots \\
&\text{for} \{ I_n = 1..n_n \} \\
&S(I_1, I_2, ...I_n); \\
\end{align*}
\]

This loop is represented as $L = (I_1, I_2, \ldots, I_n)(S)$ where $S$ represents the statements in the loop body and column vector $I = [I_1 \ldots I_n]^T$, with $1 \leq I_i \leq N_k; i = 1..n$, contains the loop indices.

Two iterations $S(I)$ and $S(J)$ are dependent, $S(I)\delta S(J)$, if both iterations access the same location, and at least one iteration writes into that location. Suppose iteration $S(I)$ precedes iteration $S(J)$ in lexicographical order (i.e. the execution order of the program). Several dependence relations are defined. If iteration $S(J)$ writes data which is used by iteration $S(I)$, then $S(J)$ is data dependent on $S(I)$. If iteration $S(J)$ writes data into the same location that is read by iteration $S(I)$, then $S(J)$ is anti-dependent on $S(I)$. If iterations $S(J)$ and $S(I)$ write into the same location, then $S(J)$ is output dependent on $S(I)$.

The dependence distance vector between two dependent iterations $S(I)$ and $S(J)$ is defined as

\[ d = J - I \]

For each pair of array elements where one is a write, the dependence distance vector is the distance between the two elements. E.g. the dependence distance vectors of a loop with statement


are $d_1 = [4, -2]^T$ and $d_2 = [4, 2]^T$. The dependence matrix $D$ contains the dependence distance vectors of the loop, i.e. $D = [d_1 \ldots d_m]$ where $m$ is the number of dependence distance vectors.

In many loops, the dependence between the array indices is fixed and the corresponding dependence matrix is constant. Loops with a constant dependence matrix have uniform dependencies. In these loops the array indices have the form $i + c$ where $c$ is a constant. A canonical form of a uniform loop nest is:
The loop variables $S$ in iterations $I$ are functions of the loop variables $S$ in iterations $I - d_1, \ldots, I - d_m$. The dependencies between the iterations are expressed by the index expressions $I - d_i$, i.e. iteration $I$ depends on iterations $I - d_1, I - d_2 \ldots I - d_m$.

Consider dependence matrix $D = [d_1 \ldots d_m]$. Then the dependence relations satisfy the following lemmas:

**Lemma 1** Two index points $I_1$ and $I_2$ are dependent if and only if

$$I_1 - I_2 = Dy, \quad y = (y_1 y_2 \ldots y_m)^T \in \mathbb{Z}^m$$

Equation (1) defines a subset of dependent iterations. A loop is partitioned in a number of subsets. Every subset is executable in parallel, but the iterations in each subset need to be executed sequentially. A few questions arise:

- How are the parallel sets identified?
- What loop transformation is needed to execute the sets in parallel?

Both issues are solved by a unimodular transformation of the dependence matrix, followed by a regeneration of the loops traversal in lexicographical order.

### 4.2. Unimodular transformation

A **unimodular matrix** is a square matrix of integer elements with a determinant value of +1 or -1. A useful property of a unimodular matrix is that its inverse is also unimodular. Likewise, the product of two unimodular matrices is also unimodular.

**Definition 1** Unimodular transformation of the index set

A uniform transformation of the index set $I \in \mathbb{Z}^n$ is the index set $Y = U I$ with $U_{n \times n}$ a unimodular matrix.

Unimodular transformations are of interest for constructing parallel partitions because:

1. There is a 1-1 mapping between the index sets $I$ and $Y = UI$. 
2. The dependencies remain invariant after a unimodular transformation, i.e.

\[ I_1\delta(D)I_2 \Leftrightarrow UI_1\delta(U\!D)UI_2. \]

where \( \delta(D) \) and \( \delta(U\!D) \) denote dependencies with respect to dependence matrix \( D \) and \( U\!D \) respectively.

3. There exist algorithms to convert a dependence matrix \( D \) to a triangular or diagonal form.

4. The maximal parallel partitions of triangular or diagonal matrices is known.

The algorithm to transform a dependence matrix into unimodular form is based on an ordered sequence of elementary (unimodular) row- or column-operations:

- exchange two rows;
- multiply a row or column by -1;
- add an integer multiple of one row or column to another row or column.

The elementary transformations are obtained by a left (for rows) or right (for columns) multiplication with a unimodular matrix representing an elementary operation. Multiplication of the elementary matrices yields the unimodular transformation matrices \( U \) and \( V \) such that \( D^t = DV \) or \( D^d = UDV \) with \( D^t \) and \( D^d \) the conversion of the dependence matrix \( D \) into triangular or diagonal form. The algorithms for these transformations are given in [12].

**Lemma 3** The triangular dependence matrix \( D^t = DV \) with \( V \) unimodular, represents the same dependencies as the dependence matrix \( D \)

Proof. With \( D = D^tV^{-1} \) the dependence equation (2) of two dependent iterations \( I_1 \) and \( I_2 \) becomes

\[ I_1 - I_2 = D^tV^{-1}y, \quad y \in \mathbb{Z}^m \]

Now, let \( y' = V^{-1}y \), then

\[ I_1 - I_2 = D^ty', \quad y' \in \mathbb{Z}^m \]

which expresses that \( I_1 \) and \( I_2 \) are dependent under dependence matrix \( D^t \). \qed

As a result, it is possible to parallelize the loops and calculate the new loop boundaries using the triangular dependence matrix.

4.3. Loop partitioning

With a triangular dependence matrix \( D^t = DV \), obtained by a unimodular transformation of the dependence matrix \( D \), the dependence set, i.e. the iterations dependent on a particular iteration \( I_0 \) are given, using equation (2):
\[ I_1 = I_{0,1} + D^t_{11} y_1 \]
\[ I_2 = I_{0,2} + D^t_{21} y_1 + D^t_{22} y_2 \]
\[ \ldots \]
\[ I_n = I_{0,n} + D^t_{n1} y_1 + \ldots + D^t_{nn} y_n \]

Now, for an arbitrary iteration \( I \), a unique label \( I_0 = [I_{0,1}, I_{0,2}, I_{0,3}] \) with \( I_{0,i} \in \{0, \ldots, D^t_{ii} - 1\} \), is calculated as follows:

\[ I_{0,1} = I_1 \mod D^t_{11} \]
\[ y_1 = (I_1 - I_{0,1})/D^t_{11} \]
\[ I_{0,2} = (I_2 - D^t_{21} y_1) \mod D^t_{22} \]
\[ y_2 = (I_2 - I_{0,2} - D^t_{21} y_1)/D^t_{22} \]
\[ \ldots \]
\[ I_{0,i} = (I_i - \sum_{j=1}^{i-1} D^t_{ij}) \mod D^t_{ii} \]
\[ y_i = (I_i - I_{0,i} - \sum_{j=1}^{i-1} D^t_{ij})/D^t_{ii} \]

Each label generates a different set of dependent iterations, using equation (2). The number of different labels, and therefore the parallelism \(|L|\) of the loop \( L \) is

\[ |L| = \prod_{i=1}^{n} D^t_{ii} = |\det(D^t)| = |\det(D)|, \]

since the unimodular transformation \( D^t = DV \) maintains the absolute value of the determinant \( \det(D) \).

### 4.4. Example

Consider the following loop:

```plaintext
for \( I_1 = 1..16 \)
  for \( I_2 = 1..16 \)
    for \( I_3 = 1..16 \)
      \( a(I_1, I_2, I_3) = A(I_1 - 1, I_2 + 2, I_3 - 4) + A(I_1 - 2, I_2 - 4, I_3 + 1) \)
      \( -A(I_1, I_2 - 4, I_3 - 2); \)
```

A unimodular transformation of the dependence matrix \( D \)
\[
D = \begin{bmatrix}
1 & 2 & 0 \\
-2 & 4 & 4 \\
4 & -1 & 2
\end{bmatrix}
\]

yields \(D^t = DV\) with

\[
D^t = \begin{bmatrix}
1 & 0 & 0 \\
-2 & 4 & 0 \\
4 & 2 & 13
\end{bmatrix}
\quad \text{and} \quad
V = \begin{bmatrix}
1 & 0 & 2 \\
0 & 0 & -1 \\
0 & 1 & 2
\end{bmatrix}
\]

Now the loop can be refactored as follows:

1. Construct parallel outer loops, generating a label \(I_0\) for each parallel dependence set. The step size of the outer loops equals the diagonal elements of \(D^t\).
2. Construct sequential inner loops, which generate the dependent iterations for label \(I_0\).
3. Maintain the body of the original loop.

This gives:

\[
\begin{align*}
\text{for}\ \forall \{I_{0,1} = 1..1\} \\
\text{for}\ \forall \{I_{0,2} = 1..4\} \\
\text{for}\ \forall \{I_{0,3} = 1..13\} \\
\text{for}\ \{I_1 = I_{0,1}..16, \text{step} \ 1\} \\
& y_1 = (I_1 - I_{0,1})/1 \\
& I_{2,\text{min}} = 1 + (I_{0,2} - 1 - 2 \times y_1) \mod 4 \\
\text{for}\ \{I_1 = I_{2,\text{min}}..16, \text{step} \ 4\} \\
& y_2 = (I_2 - I_{0,2} + 2 \times y_1)/4 \\
& I_{3,\text{min}} = 1 + (I_{0,3} - 1 + 4 \times y_1 + 2 \times y_2) \mod 13 \\
\text{for}\ \{I_3 = I_{3,\text{min}}..16, \text{step} \ 13\} \\
& A(I_1, I_2, I_3) = A(I_1 - 1, I_2 + 2, I_3 - 4) + A(I_1 - 2, I_2 - 4, I_3 + 1) \\
& \quad - A(I_1, I_2 - 4, I_3 - 2);
\end{align*}
\]

There are 52 parallel iterations in the three outer loops. Each parallel iteration consists of three sequential loops.

5. Program transformations for locality

Since data movement in FPGAs is expensive, it is important to ensure that the calculations on the same data are conglomerated. The data locality is measured by the reuse distance:

**Definition 2** Reuse distance

The reuse distance is defined as the number of distinct data elements accessed between the use and the reuse of the same data [15].
E.g. given an access pattern $A(1), A(3), A(5), A(1) \ldots$ the reuse distance of $A(1)$ is 3.

When the reuse distance is smaller than the available memory, no reload of the data is necessary. The unimodular loop transformation has a beneficial effect on the data locality, because related iterations are conglomerated in a number of serial loops. For the example in section 4.4, the reuse distance before and after the transformation is shown in figure 2.

![Figure 2. Reuse distance frequency in the example program before and after a unimodular transformation in a log-log scale. Squares are reuses of the original program, diamonds represent reuses after the unimodular transformation. The program transformation reduces the reuse distance almost 50 times.](image)

The maximum reuse distance of the original and transformed program is respectively 2298 and 47. The unimodular transformation therefore decreased the reuse distance by a factor of almost 50.

The execution of the loops in parallel assumes that the participating FPGAs have a parallel access to a large shared memory. Moreover, the loop control and the modulo calculations shown in the example 4.4 create some overhead, which hampers the efficiency. In a hardware/software codesign configuration, the FPGAs are connected to the processor via a streams channel. A further unimodular transformation allows to get rid of the modulo operation and stream the data to the different cooperating FPGAs.

5.1. Data streaming using unimodular transformations

Consider a unimodular transformation by which the dependence matrix is brought into the diagonal form: $D^d = UDV$. The dependence equation (2) becomes

$$UI = UI_0 + UD^d y', \quad y' \in \mathbb{Z}^m,$$

considering that $UDy = UDVV^{-1}y = D^d y'$ with $y' = V^{-1}y$ an arbitrary integer vector in $\mathbb{Z}^m$. Let $Y = UI$ be an iteration in the transformed iteration space and denote $Y_0 = UI_0$ the label of the dependence set $Y_0 + D^d y'$. Since dependent iterations $Y$ differ...
by a multiple of the diagonal elements, a unique label for the dependence set of \( Y \) is \( Y_0 \) such that

\[
Y_{0,i} = Y_i \mod D_{ii}, \ i = 1 \ldots m. \tag{4}
\]

It now becomes possible to assign an iteration \( I \) to a processing element as follows:

1. calculate the transformed index, \( Y = UI \),
2. calculate the iteration label \( Y_0 \) according to equation (4),
3. assign the iteration to a processor according to the formula:

\[
p = 1 + (Y_{0,1} + D_{11}^d Y_{0,2} + D_{22}^d (Y_{0,3} + \ldots + D_{n-1,n-1}^d (Y_{0,n} \ldots)))) \mod p_{\text{max}} \tag{5}
\]

where \( p_{\text{max}} \) is the actual number of FPGA processing elements available for the computation. Now in a hardware/software codesign environment, the processor executing the parallel loops, calculates the iteration label and processor number \( p \) and sends the data of the sequential loops to the streaming channel of FPGA processing element \( p \), see figure 3.

![Figure 3](image.png)

**Figure 3.** Streams operation: the parallel loops are assigned to FPGAs using formula 5.

The lexicographical ordering is maintained and each FPGA operates in a pipelined fashion on the incoming data stream.

### 6. Program transformations enhancing locality

When unimodular techniques described in the previous section are not applicable, the data locality in loops may still significantly be improved by a number of well known loop transformations.
6.1. Loop fusion

When the same data is used in adjacent loops, it is useful to merge the loops such that the data is traversed only once. E.g. the reuse distance of \( A[i] \) drops from \( M \) to 0 in the following loops:

\[
\begin{align*}
&\text{for}\{i = 1..M\} \\
&\quad A[i] + t... \\
&\text{for}\{j = 1..M\} \\
&\quad A[j] + v... \\
&\text{for}\{i = 1..M\} \\
&\quad A[i] + t... \\
&\text{for}\{j = 1..M\} \\
&\quad A[i] + v... \\
\end{align*}
\]

6.2. Loop interchange

When data is reused between two iterations of an outer loop, the reuses can be coalesced by interchanging the outer and the inner loop. In the following example, this decreases the reuse distance from \( N \) to 0.

\[
\begin{align*}
&\text{for}\{i = 1..M\} \\
&\quad \text{for}\{j = 1..N\} \\
&\quad \quad A[j]... \\
&\text{for}\{i = 1..N\} \\
&\quad \text{for}\{j = 1..M\} \\
&\quad \quad A[j]... \\
\end{align*}
\]

6.3. Loop tiling

When data is reused both in the inner and the outer loop, loop interchange is not helpful, because it will improve the inner loop accesses, but worsen the outer loop accesses. Loop tiling improves the locality in both loops by an extra loop, minimizing the data access in the inner loops to a small fraction of the data set.

\[
\begin{align*}
&\text{for}\{i = 1..M\} \\
&\quad \text{for}\{j = 1..N\} \\
&\quad \quad A[i] + B[j]... \\
&\text{for}\{t = 1..N, \text{step} \ 10\} \\
&\quad \text{for}\{i = 1..M\} \\
&\quad \quad \text{for}\{j = t..\min(t + 10, N)\} \\
&\quad \quad \quad A[i] + B[j]... \\
\end{align*}
\]

6.4. SLO: suggestions for data locality

In complex programs it is often not easy to find the parts with poor data locality, and even more so to find the right program transformation which improves the locality. E.g. current cache profilers are able to indicate the region where cache misses occur, but this is usually not the place where a program transformation improves the locality.

One of the reasons is that the cache profilers point at the endpoint of a long use/reuse chain, i.e. the cache miss. A new locality profiler, SLO (Suggestions for Locality Optimizations) [4,5] measures the reuse distance and registers a miss when the reuse distance
exceeds the memory size. At the same time, SLO analyzes the code executed between use and reuse, and finds the loops where a transformation will have the highest impact on the reuse distance. Next, a suggestion is given to fuse, interchange or tile these loops.

Since SLO is based only on the reuse distance, the profiling is independent of the memory architecture and the suggestions can be applied equally well to optimize the use of FPGA memory. In [11] SLO has been successfully applied to find and refactor loops of a 2-dimensional inverse discrete wavelet transformation (2D-IDWT) used in an FPGA-based video decoder.

7. Optimizing data reuse with smart buffers

Many C to VHDL compilers, e.g. Streams-C, Impulse C or Handel-C apply the streams paradigm to use the FPGA as a hardware procedure. For example, in Impulse C, a procedure is first tested locally for correctness, and next the hardware version of the procedure is generated for implementation on the FPGA. Software and hardware stubs are created for streaming the data to and from the FPGA. Many multimedia and signal processing applications are suitable for streaming execution, in which a signal or an image is sent to the FPGA and results are fed back. In the FPGA, the data is stored in block RAM. In the previous section, it was shown that the data locality can be optimized by suitable transformations. An additional optimization for streaming applications is the use of a smart buffer [19]. A smart buffer is a number of named registers equal to the number of variables used in the computation kernel of the application. The idea is to organize the data in such a way that they are fetched only once from the block RAM, and used repeatedly in the smart buffer for all iterations operating on the fetched data. Consider for example the following N-tap FIR filter program:

\[
S = 0 \\
\text{for}\{i = 0; i < N; i++\} \\
S = S + A[n - i] \times \text{coeff}[i] \\
Out[n] = S
\]

With N=5, the array element \(A[n]\) is used in five consecutive iterations to calculate \(Out[n], Out[n + 1], \ldots, Out[n + 4]\). A C to VHDL compiler will fetch \(A[n]\) five times from block RAM. In fact, five elements of the compute kernel are fetched from block RAM in each iteration. This creates a large overhead. By storing the elements in the registers, the access is much faster, and only one new element from block RAM is needed per iteration. This however requires the rewriting of the loop kernel computations five times, and replacing the array elements with named registers.

Since the block RAM access is the slowest operation, the use of a smart buffer achieves an almost fivefold speedup. In [14], the code produced by the Impulse C compiler has been adapted for use of a smart buffer, both for one dimensional and two dimensional applications.

In two dimensional applications such as image processing, the smart buffer consists of a window of registers representing the pixels used in the calculation. For example, in edge detection, the window consists of the pixels surrounding pixel \(A[i][j]\). A two dimensional smart buffer creates an additional complication in that there are no results in each time step, because three data elements have to be fetched before a new result
is available. This delay can be shortened by unrolling the inner loop in such a way that two or three adjacent rows are treated in the same unrolled iteration. In [14] it is shown that the application of the smart buffer technique on a FIR filter and an edge detection algorithm yield a speedup of respectively 4.14 and 4.99, using a small Spartan-3 FPGA board.

8. Conclusion

The use of FPGAs for high performance computing has drawn a lot of attention in the scientific world, at different levels of the algorithmic, application and hardware side. The computer architects have recognized the usefulness of FPGAs, as is evidenced by supercomputing companies like Cray and Silicon Graphics, which have successfully embedded FPGAs in their systems [9,34]. In the academic world, the paradigms by which FPGAs can be used have shown that there are plenty of opportunities [13]. Even so, we are still looking for one consolidating paradigm which tries to synthesize the findings of different approaches. Finally, the compilers need to be aware of specialized program transformations that deal with the characteristics of FPGAs, notably the slower processing clock, the limited amount of block RAM memory and the implications of the streaming communication, which seems to be the prevailing connection between processor and FPGA. In any case, FPGAs and reconfigurable computing will remain a very important building block for parallel and high-performance computing.

References


Nondeterministic Coordination using S-Net

Alex SHAFARENKO

Abstract. This paper presents the results obtained in the project S-Net by members of the Compilation Technology and Computer Architecture (CTCA) group at University of Hertfordshire, U.K. We argue that globally distributed HPC will require tools for coordination of asynchronous networked components, and that such coordination can be achieved by reducing the vertex in- and out-degrees of the processing nodes to 1, using single-input single-output combinators for network construction and by externalising the component state. This approach is presented first as a set of language design principles and then in the form of coordination language. The language is illustrated by an application example.

Keywords. Asynchronous distributed computing, coordination languages, stream processing.

Introduction

The subject of this paper is coordination of component networks. The term coordination was proposed in [2] to describe the double-layer model of distributed computing where the upper layer is populated with sequential computational modules that send and receive data across an interface with the lower layer, which supports communication and concurrency requirements of the module while being free from any computational concerns. It was believed that these layers can be kept fairly separate, with the upper layer being application-specific and the lower one a kind of distributed-processing virtual machine with a large degree of generality and portability. A classical example of this approach is the coordination language Linda [2], which is not really a language but a standardised extension of any imperative language. The extension introduces a few communication/concurrency primitives, which can be implemented even as library functions without any change in the host language itself (but that, under most circumstances, would not be efficient).

Why have coordination languages not been successful? Why does one still use message-passing libraries in high-performance computing, such as MPI? Part of the answer is that coordination never managed to detach itself from the computational language completely, and if it is not completely separate, componentisation is difficult to achieve. Indeed, in a large application, the information about module interactions via coordination

1Corresponding Author: Prof Alex Shafarenko, CTCA Group, Department of Computer Science, University of Hertfordshire AL10 9AB, U.K.; E-mail: a.shafarenko@herts.ac.uk
primitives is as much part of the application code as its other control structures, such as loop control or general condition logic. Coordination at this level degenerates to specific instructions that implement instances of communication and concurrency management, providing neither separate enough quality, nor sufficient abstraction to merit an additional programming language with its new concepts, syntax, cost intuition, etc. It is quite understandable therefore that instead of Linda-extensions of Fortran, computational scientists use Fortran with MPI; this combination is well understood and appears fit for the purpose – at the level of abstraction that such tools engender.

In the project S-Net[10]we set ourselves the goal to separate coordination and computation languages completely. It is only possible when components are completely encapsulated and when any linkages between them take place via an abstract component interface defined and described exclusively in the component language semantics. The only way not to mix up the control structures of the modules with the concurrency and communication control of their environment is simply to have none of the former exposed to the latter. We know of no better way to do this than to demand that computational components have no persistent state, i.e. that they act as a function that receives a message and responds to it with some output messages after which its state is initialised again. The environment would consequently not need to know anything about the innards of the components, nor would the components need to know about the state of the environment in the brief moment between receiving a message and responding to it; all the component requires to perform its entire computation should be contained in the message, the separation thus being complete. As a result, the computational language needs a simple input/output interface for receiving messages (always a blocked receive since there is no persistent state and hence nothing to do until the data arrives), and a primitive for a message send, which can be considered non-blocking, since the fact that the outgoing message has not been received yet cannot influence the actions of the component as no further message input into it will follow before re-initialisation. The input interface, i.e. the blocking message receive, can even be implemented via the component’s function parameter list, without the need for a library extension. The output interface for multiple message sends would normally require just one external function in the supporting library – and that is the extent of what the component designer needs to known about the coordination layer and its tools to be able to use our approach.

The glue that combines such components together is a network of streams, dynamically managed by a program written exclusively in a coordination language. The language should describe the connection of components into a network; the aggregation and de-aggregation of data objects into/from messages, splitting messages into further messages and synchronising a few of them into a single one (bearing in mind that a computational component cannot do the latter as it receives messages one at a time and is not allowed to have a persistent state), as well as initiating network extension (reification of component and network instances), network retraction (when parts of it become underused) and possibly reconfiguration. Since high-performance computing finds itself in an age of globally distributed and heterogeneous systems, the project S-Net focuses on asynchronous coordination, which is known to the HPC community under the name ‘one-sided communication’, or message-driven computing.

The idea of an asynchronous network of processes that co-operate to perform a collective computation goes back to Kahn [6] and the mid-70s. Kahn proposed a view on parallel computing as a stream processing network with infinite buffering, at the nodes
Figure 1. An example Kahn network. Single dashed lines represent streams between components and double ones input and output to the whole network.

of which computational functions input messages from several edges (input streams) and send messages along several other edges (output streams) when and as their computations progress, see fig 1. Kahn’s motivation came from the desire to represent parallel computing semantics, and indeed networks of this type have a remarkable property: no matter in what order the functions are allowed to compute and at what speed, the output streams of the entire network carry the same sequences of messages. The attraction of Kahn’s approach is thus in its freedom from the details of component behaviour. The programmer is liberated from concerns such as deadlock, race conditions, etc, and the price paid for that is the self-containment of the processing functions: they only interact with each other along their connecting streams; no direct access to each other’s state or the state of the environment is permitted. There is also the requirement that the processing functions must be causal, i.e. they cannot retract messages already sent when they receive further data, but in practice this is hardly a restriction.

Interest to Kahn networks as a semantic model is continuing, but the practical application of streaming networks has been largely confined to synchronous systems, such as Lustre[5], Esterel[1], etc., and that was mainly in the area of signal processing. More recent additions to the family of stream-processing languages is StreamIt[8], which is based on Java, but supports structured hierarchical networks, and Eden[7], based on Haskel and using lazy lists as streams. With the proliferation of global computing, such as Grid and Cloud, asynchronous processing will become more and more important. On the other hand, issues of inter-institutional resource sharing make it especially critical that software is presented in reusable form, i.e. components and an appropriate technology for their linkage and coordination. It is our position that componental solutions close to Kahn networks, but better structured and enhanced with modern type systems are likely to facilitate global computing and efficient software development in HPC. Our goal being to address the issues of asynchronous coordination of components in a dynamic streaming network, the rest of the paper is organised as follows. First we will look at structural challenges of representing a streaming network in a programming language: the graph-topology challenge and the feedback challenge. We will propose some remedies that make algebraic description of networked processing relatively easy and state them in the form of design principles. Next we will focus on the structural primitives that support network definitions: combinators and synchronisers, which will lead us to a brief overview of the coordination language S-Net. In it we will touch upon the issues of type and inheritance, since types are used in S-Net for routing purposes, which lightens up the notation considerably, and since inheritance is an important mechanism that ensures flexibility in component aggregation thus further improving the genericity of the combinators. Finally we will devote a section to a small example of S-Net programming.
1. Streaming networks: structural challenges and proposed design principles

For a stream-processing language the variety of network topologies and the need to encode them in a program in a clear and expressive way constitute a major design challenge. Works by Stefanescu [12] provided a classification of communication graphs for stream processing, and suggested some structuring primitives. However, the formalisation remains quite complex and not suitable for much more than semantics research. There are two main causes for this complexity. The one presenting the easier challenge is the multiplicity of input/output streams incident to a component. To describe the connection of such components in a programming language in a structured fashion would require a nomenclature of primitives, for example the network algebra [11], augmented with some type theory for stream types and processing functions. The algebra includes so-called branching constants, which are primitives that describe the connection patterns of a pair of multi-stream components. It is hard to visualise how exactly components are connected when faced with a sizeable formula saturated with branching constants.

The other cause, which goes deeper than any concerns of expressivity, is the fact that a general streaming network contains cycles. The network algebra offers a primitive to support cycles, but to reason about cyclic processing is as hard as it is to reason about a large set of functions recursively calling one another on a non-recursive data structure (a stream sequence in this case). Besides that, a cyclic network is prone to deadlock unless measures are taken to schedule the activities in a safe way, which is not always straightforward.

1.1. Acyclic networks

Let us consider the first complication identified above. To start with, is the multiplicity of input/output streams to a component strictly necessary? Could there be several logical input or output streams sharing the same sequence of messages? Answers to these questions depend entirely on the buffering mechanism underlying the communication abstraction. Clearly if the buffering space is “sufficient” (and the meaning of this should be defined rigourously), messages can be transparently floated into the same output stream and then demultiplexed at the other end according to their destinations. If, on the other hand, streams have their allocated buffer pools and the allocation is fixed, there is a marked difference between a singly and multiply connected parts: in the former case the transmission on an individual stream can be blocked when its buffer space is exhausted, while in the latter, this applies to the actual connection as a whole and not individual virtual streams. As a result, one virtual stream can be active enough to exhaust the buffer space of the connection and thus prevent other virtual streams from carrying messages for a long time, see fig 2. Similarly to any case of spurious dependencies that are due to the finiteness of the resources, this one can cause a deadlock if not controlled properly.

In non-real-time, high-performance streaming applications, the programmer’s intuitive view of network processing includes the notion of back pressure, i.e. the blocking of message producers when the consumer is busy. Back pressure tends to propagate back in a pipeline until it reaches the network inputs. It effectively ensures that each cross-section of a pipeline operates with the speed of the slowest stage, and, given static per-stream buffer allocation in the absence of cycles, programming with back pressure guarantees progress. In managing the concurrency of asynchronous networks, back pressure alone
is not usually sufficient. Even in an acyclic graph (which is similar to a pipeline) there are multiple streams flowing across the network, and also any synchronisation points depend on more than one stream for progress. For that reason, the coordination programmer must introduce problem-specific mechanisms for throttling concurrency in order to avoid hazards such as starvation and deadlock. But then it should be possible to use the same mechanism to ensure that the top and bottom cases in fig. 2 have similar behaviours. The bottom case, though, is much easier to manage in terms of network composition, as we shall see below, leading to a design principle:

**Principle 1 (SISO)** Every structural unit of a network from the whole network down to the individual components must have a Single Input and a Single Output stream.

For all its deceptive simplicity, the stream aggregation exemplified in fig. 2 is non-trivial. Indeed the form of coordination that it requires is known as a stream merger. Even though this is a service provided to a component by the coordination layer, it is not necessarily transparent to the recipient. Indeed the situation in the figure can be refined by showing the part on the left as being a composition of two chunks and assigning each virtual stream to either chunk. The operation of the chunks could be completely independent, yet the single actual stream out of the left-hand part combines messages from the chunks in some order. The right-hand part may require messages from both chunks for progress, and it may need them in a different order. In the multiply connected case, the correct order could be achieved easily, by blocking a stream until messages from it are required. By contrast, if we follow the SISO principle, we must also provide a reordering mechanism so that a consumer may consume in the order of need, rather than in the arbitrary order of the merger.

The trick here is to avoid having to specify either order in the language, and to rely on the adaptivity of the implementation. Looking at it from the coordination language point of view, we arrive at the following principle:

**Principle 2 (nondeterminism)** The confluence of streams can be in no particular order; when that is the case, (i) it should be clearly indicated in the coordination program and (ii) the implementation is expected to monitor the recipient of the joint stream and change the priorities of the stream merger to save buffer space and reduce the processing latency.

Part (ii) of the principle refers to the use of blocking and back pressure not dissimilar to the multiply connected case of fig. 2, which makes the combination of SISO and nondeterminism almost the replacement of multiply connected networks.

Staying within the confines of an acyclic network, the most general topology of a fully connected network is a directed acyclic graph (DAG). The in- and out-degrees of each vertex correspond to the number of input and output streams of the component located at the vertex. To satisfy the SISO principle, first of all let us augment the graph with a global input node \( \text{In} \), into which the confluence of all input streams flows as a single combined stream \( \alpha \). The node \( \text{In} \) is (multiply) connected to the nodes that must receive an input stream and its function is to select the relevant portion of \( \alpha \) and deliver it to the relevant node. Also augment the graph with a node \( \text{Out} \) that takes all the global output streams and flows then into a single combined output stream.
Next, for each component $c_i$ compute $s_i = \max \pi_{i,0}$, which is the maximum path length between $c_i$ and $c_0 = \text{In}$. Then arrange the components in a serial-parallel composition as shown in fig. 3 in the ascending order of their $s_i$, from 0 to some $N = s_k$, where $c_k = \text{Out}$. Here a triple line at the bottom of a parallel group represents the bypass stream that carries messages not addressed to the other components of the group. Finally drop the In/Out nodes as they have become redundant. The SISO version of the graph (shown at the bottom of the diagram) would be incomplete without some routing rules that define which member of a parallel group should receive which incoming message. Since routing is essentially matching the beginning of a path with its end, it is profitable to use a type system for that; this way, other type constraints that component interfaces may export could be captured by the same mechanism. The fact that type systems ordinarily deal with value properties of data rather than the topological matching of a path should not discourage us: as evidenced by algebraic data types, type systems have the ability to introduce abstract labels, which can be used for targeting specific components’ inputs.

The example in fig. 3 serves as an illustration of the following design principle:
Principle 3 Acyclic segments of the network should be coordinated as groups of subnetworks under serial and parallel composition. This provides skeletal routing; the precise routing information is to be captured by message and component types.

Of course Principle 3, if understood literally, could be construed as the directive to cascade messages through groups of components arranged in parallel. That is not necessarily the case. Indeed the structuring introduced by the Principle is intended for the purposes of a programming (coordination) language and is there to represent, with the assistance of an appropriate type system, the topological properties of the original multiply-connected network. The implementation can easily reconstruct that network and determine for every message type its destination for a direct dispatch. On the other hand, certain types of hardware (e.g. massively parallel multicore processors) do not allow arbitrary connectivity anyway, so having to cascade messages through a chain of routers may not be an extra burden.

1.2. Cyclicity

Practical networks tend to be cyclic. Indeed any network solution that involves iteration must apply the same algorithm to data several times, and in an acyclic network that would result in node duplication along with the undesirable duplication of the components placed at the nodes. Yet, for reasons mentioned earlier, it would be beneficial to avoid cyclic configurations in a coordination language. Under normal circumstances these requirements would seem irreconcilable; however for streaming networks there is at least a compromise solution, which we will consider next.

It is true that a cyclic network is not equivalent to any finite acyclic network. However, if we allow for infinite networks then cyclicity is quite avoidable. Indeed, a cyclic graph can be unrolled by repeatedly following the edges that form a cycle and duplicating the vertices that have already been visited ad infinitum. Doing this for every cycle that occurs in the graph will convert it to an infinite regular, acyclic graph. Informally, a feedback loop is being replaced by a feed-forward infinite pipeline, see fig. 4. Vertex duplication is, of course, predicated on the fact that the components located at the original and copy vertices can be made identical. This, in turn, requires them to be stateless, since otherwise it would be possible to find the original component and its copy in different states and detect the difference between the cyclic and unrolled configurations. Feed-forward networks are a useful abstraction in its own right: they can represent finite, repetitive, pipelined computations even of a stateful network, if the amount of unrolling is limited (cf. loop unrolling in code optimisation) and if the state information can be decoupled from the component and communicated over the pipeline alongside other data. If
a feed-forward structure is used to represent cyclicity, the key difference between them, as made clear in fig. 4, is the delivery of the input stream. In the cyclic configuration the input messages and the feedback stream arrive at the input of a single subnet $A$, while in the unrolled version the input stream has to be forwarded to the $k$th generation replica, with ever increasing $k$. The forwarding should be the responsibility of $A$; however, to avoid the potentially inefficient cascade it is best to use the coordination language facilities that are required already for bypassing messages in an acyclic network, as shown in fig. 3. The coordination language compiler will then have a chance to recognise cascaded forwarding and to generate management code that eliminates it. Another optimisation the compiler or the run-time system may need to support is the management of the chain length. Indeed, as new messages enter the chain, the replicas of $A$ will generally produce records that are diverted down to the output stream and records that continue to the next replica. It is reasonable to assume that at some point $k = k_t$ the replica $c_{k_t}$ will not produce any output for the next one and so the chain will stop expanding. For each new message entering the chain the value of $t$ will generally be different, but when $t$ decreases, $t$ may be expedient to collect the tail replicas as garbage (assuming that any persistent state that they may have accumulated has been used up and destroyed\footnote{It should be noted that although application components in our approach have no persistent state, coordination objects generally do, but that state is visible to the coordination layer.}). To summarise, here is the next design principle:

**Principle 4** To represent network cycles and repeatable computations, introduce a feed-forward pattern whereby a single subnet is replicated conceptually infinitely, with only a finite part being used at any given time. Output is achieved by flowing messages of the output type of $A$ into a single stream as shown in fig. 4 and the input can either be consumed by the first replica or cascaded by replicas together with other continuation data. The coordination compiler and its runtime system must strive to recognise and eliminate cascades and inactive replicas.

### 1.3. Index-parallelism

Spatial decomposition is an important source of parallelism and the SPMD pattern of computing is essential for networked computing. In the context of stream processing spatial decomposition can be thought of (and the corresponding network combinator represented) as a parallel composition of a number of replicas, with the input data stream split between them based on an index contained in each message, see fig 5. The outputs of the replicas will need to be flowed together into a single output stream to satisfy the SISO principle. The difficulty in providing the index-based split is in that it breaks the opacity of the message; messages send to the pattern in question must be readable by the coordination layer, whereas up to now we have not needed access to any values of message data for coordination. This gives rise to the following

**Principle 5** The type system should allow messages to contain named fields. Some of these fields should be transparent for coordination purposes, others are opaque and can only be processed by components. The transparent fields need to contain only integer data. The type system must ensure that fields not intended for coordination are never read by the coordination layer.
1.4. Synchronisation

The SISO principle combined with the lack of persistent state makes it impossible for a component to use data from more than one message. However, there is nothing unusual in a design in which two subnetworks prepare and send messages to a third one, which in turn must combine the messages first and then do some processing on joint data. The solution is to introduce a synchronisation cell, or synchrocell for short, into the coordination layer. A synchrocell is a SISO entity that combines messages into one. It does not need to synchronise more than one group of messages since our approach already has two replication factories in the toolkit: the feed-forward pattern and an index-splitting pattern. Consequently a use-once synchrocell inside the former pattern can be employed as a synchronising queue (provided that there is adequate bypassing), and inside the latter one it will act as an index-based matching store.

One of the advantages of the disposable, use-once synchrocell is that it helps to externalise the state of a component that needs a state. Indeed, a stateful SISO component would read one message \( m \in I \) off the input stream, transition to the appropriate state \( s \in S \) defined by the current state and some state-transition function \( F : S \times I \rightarrow S \times O \), also producing some output \( o \in O \), possibly empty. Figure 6 displays a network solution that honours the principles stated earlier. Here the use-once synchrocell bypasses the second, third, etc messages of the same type, and joins the first messages of either type into a combined message. The transition component \( F \) expects a message that contains both \( inp \) and \( s \) and produces two messages in response to it: a message of type \( s \) and an optional message of type \( oup \). Messages of types other than the input type of \( F \) are bypassed. The construct inside the balloon is used as an operand to the serial replication pattern. This results in an asynchronous stateful stream-transforming network: consecutive replicas of \( F' \) store the input messages in the order of their arrival (regardless of the mergers’
nondeterminism) and the state is cascaded through the state-transition function along the spine of the replication pattern. Note that since a component is not statically constrained to a fixed number of output messages (we have stated no such principle) but only to a fixed number of message types, it takes little thought to see that $F$, by sending more than one $s$ to its output, can start a second processing thread that will steal some input and will produce independent output. By extending the signature of the transition function with more message types, it is possible to ensure that the threads coordinate work between themselves, synchronise when necessary and produce a consistently ordered output, all of that without changing the basic template of figure 6.

To summarise, here is

**Principle 6 (Synchrocell)** Stream synchronisation is to be achieved by use-once SISO synchrocells. A synchrocell must specify the types of records that it joins into one. When records of all of these types arrive to the synchrocell in any order, the cell produces the joint record. Further records of any of the types specified in the synchrocell are bypassed to the cell output. If the network requires periodic or arrayed synchronisation, then replication facilities should be used to create synchro-queues and matching stores, respectively.

2. **S-Net**

A coordination language based on the above principles was first outlined by the author of this article in 2005 as part of the EU grant proposal AETHER[9], and later the language was refined, expanded, rigourously defined and implemented by members of the CTCA group in collaboration with several external organisations. In this brief survey of S-Net we have no space for a proper discussion of implementation- and application-related issues; we will only provide an exposition of the language, largely following a recent paper [4]. Full project documentation including a complete language definition, reports on the principles and technical details of the current programming environment, including a usable compiler and a run-time library, are all available from the public web site [10].

2.1. **Types in S-Net**

Let us begin with the outline of the type system of the coordination language. It is an essential part of the language as defined by Principle 3: not only do types in S-Net ensure that messages are processed by components that correctly interpret their structures, types are also crucial in determining which route a message takes. Also, Principle 5 dictates that messages provide enough structure to be able to encapsulate data items as fields and to describe data dependencies on the network graph in terms of field occurrence in various message types. This suggests the treatment of messages as records.

2.1.1. **Record types**

The type system of S-Net is based on non-recursive variant records with record subtyping. Informally, a type in S-Net is a non-empty set of anonymous record variants separated by vertical bars. Each record variant is a possibly empty set of record entry names, enclosed in curly brackets. We distinguish two different kinds of record entries: fields
and tags. A field is characterised by its field name; it is a label associated with an opaque value at runtime. Hence, fields can only be generated, inspected or manipulated by using an appropriate component (as opposed to coordination) language. A tag is represented by a label enclosed in angular brackets. At runtime tags are associated with integer values, which are visible to the component code and the S-Netcode. Tags are used to control the matching of records as per Principle 3.

We illustrate S-Net types by a simple example from 2-dimensional geometry. A rectangle may be represented by the S-Net type

\{x, y, dx, dy\}

providing fields for the coordinates of a reference point (x and y) and edge lengths in both dimensions (dx and dy). Likewise, we may represent a circle by the center point coordinates and the radius:

\{x, y, radius\}

Using the S-Net support for variant record types we may easily define a type for geometric bodies in general, encompassing both rectangles and circles:

\{x, y, dx, dy\} | \{x, y, radius\}

Often it is convenient to name variants. In S-Net this can be done using tags:

\{<rectangle>, x, y, dx, dy\} \{<circle>, x, y, radius\}

S-Net supports type definitions; we refer the interested reader to [3] for details.

2.1.2. Record subtyping

S-Net supports structural subtyping on record types. Subtyping essentially is based on the subset relationship between sets of record entry names. Informally, a type is a subtype of another type if it has additional record entries in the variants or fewer variants. For example, the type

\{<circle>, x, y, radius, colour\}

representing coloured circles is a subtype of the previously defined type

\{<circle>, x, y, radius\}

Likewise, we may add another type to represent triangles:

\{<rectangle>, x, y, dx, dy\} \\
\| \{<circle>, x, y, radius\} \\
\| \{<triangle>, x, y, dx1, dy1, dx2, dy2\};

which again is a supertype of
as well as a supertype of

{<circle>, x, y, radius, colour} .

Our definition of record subtyping coincides with the intuitive understanding that a subtype is more specific than its supertype(s) while a supertype is more general than its subtype(s). In the first example, the subtype contains additional information concerning the geometric body (i.e. its colour) that allows us to distinguish, for instance, green circles from blue circles. In contrast, the more general supertype identifies all circles regardless of their colour. In our second example, the supertype is again more general than its subtype as it encompasses all three different geometric bodies. Subtype {<circle>, x, y, radius, colour} is more specific than its supertypes because it rules out triangles and rectangles from the set of geometric bodies covered. Unlike subtyping in object-oriented languages our definition of record subtyping is purely structural; {} (i.e. the empty record) denotes the most common supertype.

2.1.3. Type signatures

Type signatures describe the stream-to-stream transformation performed by a network. Syntactically, a type signature is a non-empty set of type mappings each relating an input type to an output type. The input type specifies the records a network accepts for processing; the output type characterises the records that the network may produce in response. For example, the type signature

\{a, b\} \mid \{c, d\} \rightarrow \{<x>\} \mid \{<y>\}, \{e\} \rightarrow \{z\}

describes a network that accepts records that either contain fields a and b or fields c and d or field e. In response to a record of the latter type the network produces records containing the field z. In all other cases, it produces records that either contain tag x or tag y.

2.1.4. Flow inheritance

Up-coercion of records upon entry to a certain network creates a subtle problem in the stream-processing context of S-Net. In an object-oriented setting, the control flow eventually returns from a method invocation that causes an up-coercion. While during the execution of the specific method the object is treated as being one of the corresponding superclass, it always retains its former state in the calling context. In a stream-processing network, however, records enter a network through its input stream and leave it through its output stream, which are both connected to different parts of the whole network. If an up-coercion results in a loss of record entries, this loss is not temporary but permanent.

The permanent loss of record entries is neither useful nor desirable. For example, we may have a network that manipulates the position of a geometric body regardless of whether it is a rectangle, circle or triangle. The associated type signature of such a network could be as simple as \{x, y\} \rightarrow \{x, y\}. The network would accept circles, rect-
angles and triangles and would process their common data (i.e. the position) and ignore their individual specific fields and tags. Obviously, we must not lose those additional entries as a consequence of the automatic up-coercion of complete geometric bodies to type \{x, y\}. Hence, we complement the up-coercion with an automatic down-coercion. More precisely, any field or tag of an incoming record that is not explicitly mentioned in the input type of a network bypasses it and is appended to any outgoing record created in response, unless that record already contains a field or tag with the same label. We call this coercion mechanism \textit{flow inheritance}.

As an example, let us assume a record \{\text{<circle>,}\text{x, y, radius}\} hits a network \{x, y\}→\{x, y\}. While fields \text{x} and \text{y} are processed by the component code, tag \text{circle} and field \text{radius} bypass the component without inspection. As they are not mentioned in the output type of the component, they are both added to any outgoing record, which consequently forms a complete specification of a circle again.

2.2. Classes of components

In S-Net, components under coordination (also called “boxes”) are divided into three classes: user-defined boxes, which are components written by a component programmer in some component or box language; filters, which are boxes similar to user-defined boxes except they only repackage (i.e. copy, rename, duplicate and delete fields/tags of) input records and for that reason they are supported directly by the S-Net language; and synchrocells, which are S-Net constructs that realise the Synchrocell Principle.

2.2.1. User-defined boxes

From the perspective of S-Net boxes are atomic building blocks for streaming networks. Boxes are declared in S-Net code using the key word \texttt{box} followed by a box name as unique identifier and a box signature enclosed in round brackets. The box signature resembles a type signature with two exceptions: we use round brackets instead of curly brackets, and we have exactly one type mapping, which has a single-variant input type. For example,

\[
\text{box } \text{foo } (( \text{a, b, < t >}) \rightarrow (\text{a, b}) \mid (\text{< t >}));
\]

declares a box named \texttt{foo}, which accepts records containing (at least) fields \text{a} and \text{b} plus a tag \text{t} and in response produces records that either contain fields \text{a} and \text{b} or tag \text{t}. It is entirely up to the box to decide how many output records it will emits and of which of the output variants they will be. This may well depend on the values of the input record entries and, hence, can only be determined at runtime. None of this information is available to the coordination language.

Box signatures use round brackets instead of curly brackets to express the fact that in box signatures, the order of the record entries does matter. (Remember that type signatures are true sets of mappings between true sets of record entries.) The order is essential to support a mapping to function parameters of some box language implementation rather than using inefficient means such as string matching of field and tag names. For example, we may want to associate the above box declaration \texttt{foo} with a C language implementation in the form of the C function \texttt{foo} shown in Fig. 7. Here the (only) S-Net API function \texttt{snetout} is used to output the values of the fields associated with the first
snet_handle_t *foo( snet_handle_t *handle,
        int *a, mytype_t *b, int t )
{
    /* some computation on a, b and t */
    snetout( handle, 1, a, b );
    /* some computation */
    snetout( handle, 2, t );
    return( handle );
}

Figure 7. Example box function implementation in C

and the second variant of the output type as per signature stated earlier. The handle
is used by the run-time system to relay to snetout the contextual information from the
coordination layer.

2.2.2. Filter boxes

The filter box in S-Net is devoted to housekeeping operations. Effectively, any operation
that does not require knowledge of field values can be expressed by this versatile built-
in box in a simple way thus making it unnecessary to produce box-language code for a
specific occasion. Among these operations are

- elimination of fields and tags from records,
- copying fields and tags,
- adding tags,
- splitting records,
- simple computations on tag values.

Syntactically, a filter box is enclosed in square brackets and consists of a type (pattern)
to the left of an arrow symbol and a semicolon-separated sequence of filter actions to the
right of the arrow symbol, for example:

\[
[ \{a,b,<t>\} \rightarrow \{a\} ; \{c=b,<u=42>\} ; \{b,<t=t+1>\} ]
\]

This filter box accepts records that contain fields a and b as well as tag t. In general,
the type-like notation to the left of the arrow symbol acts as a pattern on records; any
incoming record's type must be a subtype of the pattern type.

In response to each incoming record, the filter box produces three records on its
output stream. The specifications of these records are separated by semicolons to the
right of the arrow symbol. Outgoing records are defined in terms of the identifiers used in
the pattern. In the example, the first output record only contains the field a adopted from
the incoming record (plus all flow-inherited record entries). The second output record
contains field b from the input record, which is renamed to c. In addition there is a tag u
set to the integer value 42. The last of the three records produced contains the field b and
the tag t from the input record, where the value associated with tag t is incremented by
one. S-Net supports a simple expression language on tag values that essentially consists
of arithmetic, relational and logical operators as well as a conditional expression.
2.2.3. Synchrocells

The synchrocell is the only “stateful” box in S-Net. It also provides the only means in S-Net to combine two or more records into a single one, whereas the opposite direction, the splitting of a single record, can easily be achieved by both user-defined boxes and built-in filter boxes. Syntactically, a synchrocell consists of an at least two-element, comma-separated list of type patterns enclosed in [ ] and | ] brackets, for example

```
[ | {a, b, <t>}, {c, d, <u>} | ]
```

The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then are the records merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification: a synchrocell only accepts records that match at least one of the patterns.

The functioning of the synchrocell fully adheres to the Synchrocell Principle, including the bypass mechanism. A more subtle issue is the interplay between subtyping and flow-inheritance on the one hand and the synchrocell operation on the other. Without going into details we remark that the cell avoids multiple inheritance by only inheriting the extra entries via the first pattern; the rest of the patterns only support subtyping. This much inheritance seems both necessary and sufficient to support componental representations with one principal virtual stream; and we have not so far been able to find an application for which multiple flow inheritance would be essential.

2.3. Streaming networks

2.3.1. Network definitions

User-defined and built-in boxes form the atomic building blocks for stream processing networks; their hierarchical definition is at the core of S-Net. As a simple example of a network definition take:

```
net X {
  box foo ((a,b)->(c,d));
  box bar ((c)->(e));
}
connect foo .. bar;
```

Following the key word `net` we have the network name, in this case `X`, and an optional block of local definitions enclosed in curly brackets. This block may contain nested network definitions and box declarations. Hierarchical network definitions incur nested scopes, but in the absence of relatively free variables the scoping rules are straightforward.

A distinctive feature of S-Net is the fact that complex network topologies are not defined by some form of netlist, but by an expression language. Each network definition contains such a topology expression following the key word `connect`. Atomic expressions are made up of box and network names defined in the current scope as well as of built-in filter boxes and synchrocells. Complex expressions are inductively defined using a set of network combinators that represent the four essential construction principles.
in S-Net: serial and parallel composition of two (different) networks as well as serial and parallel replication of one network, as sketched out in Fig. 8. Note that any network composition again yields a network with exactly one input and one output stream.

**Figure 8.** Illustration of network combinators and their operational behaviour: serial composition (top-left), parallel composition (top-right), serial replication (bottom-left) and indexed parallel replication (bottom-right)

### 2.3.2. Serial composition

The binary serial combinator “..” connects the output stream of the left operand to the input stream of the right operand. The input stream of the left operand and the output stream of the right operand become those of the combined network. The serial combinator establishes computational pipelines, where records are processed through a sequence of computational steps.

In the example of Fig. 8, the two boxes `foo` and `bar` are combined into such a pipeline: all output from `foo` goes to `bar`. This example nicely demonstrates the power of flow inheritance: In fact the output type of box `foo` is not identical to the input type of box `bar`. By means of flow inheritance, any field `d` originating from box `foo` is stripped off the record before it goes into box `bar`, and any record emitted by box `bar` will have this field be added to field `e`.

In contrast to box declarations, type signatures of networks are generally inferred by the compiler. For example the inferred type signature of the network `X` in the above example is `{a,b}->{d,e}`.

### 2.3.3. Parallel composition

The binary parallel combinator “|” combines its operands in parallel. Any incoming record is sent to exactly one operand depending on its own type and the operand type signatures. The output streams of the operand networks (or boxes) are merged into a single stream, which becomes the output stream of the combined network. Fig. 8 illustrates the parallel composition of two networks `foo` and `bar` (i.e. `foo|bar`).

To be precise, any incoming record is sent to that operand network whose type signature’s input type is matched best by the record’s type. Let us assume the type signa-
ture of \texttt{foo} is \{a\} \rightarrow \{b\} and that of \texttt{bar} is \{a, c\} \rightarrow \{b, d\}. An incoming record \{a, \langle t \rangle\} would go to box \texttt{foo} because it does not match the input type of box \texttt{bar}, but thanks to record subtyping does match the input type of box \texttt{foo}. In contrast, an incoming record \{a, b, c\} would go to box \texttt{bar}. Although it actually matches both input types, the input type of box \texttt{bar} scores higher (2 matches) than the input type of box \texttt{foo} (1 match). If a record’s type matches both operand type signatures equally well, the record is non-deterministically sent to one of the operand networks.

2.3.4. Serial replication

The serial replication combinator “\*” replicates the operand network (the left operand) infinitely many times and connects the replicas by serial composition. The right operand of the combinator is a type (pattern) that specifies a termination condition. Any record whose type is a subtype of the termination type pattern (i.e. matches the pattern) is released to the combined network’s output stream.

In fact, an incoming record that matches the termination pattern right away is immediately passed to the output stream without being processed by the operand network at all. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol. Fig. 8 illustrates the operational behaviour of the star combinator for a network \texttt{foo*\langle stop\rangle}: Records travel through serially combined replicas of \texttt{foo} until they match a given type pattern, more precisely the type of the record is a record subtype of the specified type (pattern). Optionally, the exit pattern may be refined by a boolean expression on the values of the tags in the type pattern. Actual replication of the operand network is demand-driven. Hence, networks in S-Net are not static, but generally evolve dynamically, though in a restricted way.

2.3.5. Indexed parallel replication

Last but not least, the parallel replication combinator “\!” takes a network or box as its left operand and a tag as its right operand. Like the star combinator, it replicates the operand, but connects the replicas using parallel rather than serial composition. The number of replicas is conceptually infinite. Each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the given tag. Hence, all records that have the same tag value will be routed to the same replica of the operand network. Fig. 8 illustrates the operational behaviour of indexed serial replication for a network \texttt{foo!\langle T \rangle}. In analogy to serial replication, instantiation of replicas is demand-driven.

Note that this construct in combination with serial replication allows dynamic, SPMD style connections: a network such as \texttt{(A!\langle P \rangle)*\langle Y \rangle} allows \texttt{A} to receive records with a certain value of \langle P \rangle and create records with either the same or different value of \langle P \rangle which will be fed to an appropriate replica of \texttt{A}. Any output from \texttt{A} that is meant to be released should be tagged with \langle Y \rangle. It is quite obvious that dynamic communication could be made as complex as the programmer requires, but crucially the only routing issue that is dealt with dynamically is \textit{which} replica of a network a given record should be directed to, not which network, and since all replicas share the same type signature, S-Net remains type safe even under dynamic routing.
3. Code example

Having presented the language S-Net, we are ready to proceed to an example coordination program. The example that follows was developed by F. Penczek at CTCA. It is an asynchronous streaming network implementation of the classical DES encryption algorithm with boxes of very fine granularity. Anyone familiar with DES will easily recognise the computational structures involved in the solution. The key part here is \texttt{desround}, which implements one round of DES, see fig. 9. The input record for it contains the key set and and the left- and right- halves of the data block. Also present is a tag \texttt{<C>} serving as a counter. The output has exactly the same type as the input, supporting the 16 stage pipeline characteristic of the DES algorithm. Notice (see the program listing below) the presence of two splitting points and two synchro-queues: one around the Feistel network and another around the half-block swap. The asynchrony associated with nondeterministic mergers will allow different rounds of DES run in parallel for at least part of the operation, especially if the time taken by the Feistel network differs from instance to instance.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{network.png}
\caption{A graphical representation of the DES solver network}
\end{figure}

Now let us take a look at the network implementation of Feistel itself. The \texttt{netExpandAndKeySelect} network is the first stage of the Feistel pipeline. It divides the incoming stream into two, puts them through two corresponding processing boxes and then zip the results up using a synchro-queue. The rest of the pipeline performs key XORing and some substitution and permutation defined inside user boxes.

\begin{verbatim}
net des ( {Key, Pt} -> {Ct} ) {
  box xor( (Op1, Op2) -> (Result));
  box InitialP( (Pt) -> (L, R));
}
\end{verbatim}
box genSubKeys( (Key) → (KeySet) );
box KeyInvert( (KeySet) → (KeySet) );
box FinalP( (L, R) → (Ct) );

net desRound {
  net feistel {
    net ExpandAndKeySelect {
      box BitExpand( (R) → (Rx) );
      box SubKey( (KeySet, <C>) → (KeySet, NextKey, <C>) );
    }
    connect
    \[[R, KeySet, <C>] → {R}; {KeySet, <C>}\] ..
    (BitExpand
     | SubKey
    ) .. \[[l \{KeySet, NextKey, <C\}, \{Rx\} l]*
    \{Rx, KeySet, NextKey, <C\}];
  }
  net KeyMix
  connect \[[\{NextKey, Rx\} → {Op1=NextKey, Op2=Rx}\] ..
  xor .. \[[\{Result\} → {BitStr=Result}\];
  box Substitute( (BitStr) → (SStr) );
  box Pbox( (SStr) → (Rf) );
}
connect ExpandAndKeySelect .. KeyMix .. Substitute .. Pbox;

net XorHalfBlocks
connect \[[\{L, Rf\} → {Op1=L, Op2=Rf}\] .. xor
  .. \[[\{Result\} → {R=Result}\];
}
connect \[[\{L, R, KeySet, <C>\} → \{L, R, KeySet, <C=C+1\};\{Rn=R\}\] ..
  (\[[\{Rn\} → \{L=Rn\}\]
   |
   (\[[L, R, KeySet, <C>\} → \{L, KeySet, <C\}] ..
   (\[[L]→[L]\]
    | feistel
   ) .. \[[\{L\}, \{KeySet, Rf, <C\}] l]*\{L, KeySet, Rf, <C\}
   .. XorHalfBlocks
  )
  .. \[[\{L\}, \{R, KeySet, <C\}] l]*\{L, R, KeySet, <C\};
4. Conclusions and future work

Principles of asynchronous coordination of a distributed, componentised application have been presented and a language based on them outlined and discussed. The ambition of the S-Net project is to provide a tool that would eliminate “programming in the large” in a language intended for “programming in the small”. We wish to see a complete separation of two forms of distributed applications design: component and system, with the former focused on the mathematics of data processing and the latter exclusively on its logistics. S-Net is a glue that we believe will be effective in combining the two without one contaminating the other. It is a flexible glue, which is at home with high latency communications, locally encapsulated data and network-based computing in the style of Grid and Cloud. At present only a P-thread implementation of S-Net is available, but work is underway to port S-Net to MPI and its Grid-based versions, after which realistic applications will be attempted.

Two major challenges lie ahead. One, on which work has already started, is the extension of S-Net with facilities for self-reconfiguration and self-adaptation. This has been being advanced by an ongoing PhD project already and a formal semantics of S-Net, including new combinators for self-adaptation, is among the early results of the project. Self-adaptation will enable a coordination program to do computational steering, adaptive visualisation and will provide a basis for application-specific resource management and fault tolerance. An interesting feature of acyclic networks is that self-adaptation is strictly impossible without some disciplined form of feedback, such that it is not prone to deadlock, but is able to deliver self-monitoring data back upstream to trigger network reconfiguration. This raises new, interesting theoretical and implementation issues in stream processing technology.

The second challenge is to do with the dual nature of S-Net. It is, primarily, a coordination language, which means that the programmer’s intuition is to treat S-Net boxes as reified stream transformers deployed in the program memory of networked hosts. There is a competing view in which S-Net is treated as a functional language modulo non-determinism; this allows for the treatment of boxes as abstract functions that can be reified when and as necessary, S-Net connect-formulae as expressions with high-order functions, i.e. network combinators, and S-Net concurrency as one of a functional program. While not particularly fruitful for the coordination programmer, who stands to lose all cost intuitions if he or she follows this mode of thinking, it has opened up an implementation avenue based on the idea of abstract agents walking the S-Net graph and carrying messages along. It is likely that elements of graph walking and box reification will have to be combined if resource management below the S-Net level is to be linked up with efficiently.

Above all, S-Net needs users. Our immediate plans include a collaborative project with a CFD group in Novosibirsk that will re-code their suite of core algorithms as S-Net

\[
\text{connect genSubKeys } \ldots ([] \mid ([Decipher] \to []) \ldots \text{KeyInvert}) \ldots \\
\text{InitialP } \ldots ([L,R,KeySet] \to [L,R,KeySet,C=0]) \ldots \\
desRound*{C} \text{ if } <C==16> \ldots \\
\text{FinalP } \ldots ([KeySet, C] \to []);
\]
boxes, while encoding CFD applications as coordination programs. Several workshops are being planned to enable applications programmers to try S-Net. The S-Net information Web site [10] will continue to be maintained as a source of software, documentation, application examples and tutorial and workshop announcements.

Acknowledgements

We acknowledge EU Framework VI funding under the project “AETHER”, which helped to define and implement S-NET. The implementation concept, design of the first S-Net compiler and its continual refinement are due to Dr C.Grelck of CTCA, assisted by F.Penczek, also of CTCA, and our partners at VTT, Helsinki, who contributed a large amount of code. The first type system was refined and implemented by H.Cai of Imperial College, London. We acknowledge the contributions of Dr S.-B. Scholz and S. Herhut of CTCA who helped with box-language interface issues and made several important contributions in discussions about the language design. We are indebted to Thales Research, Paris, who provided a high-performance signal processing application and assisted us in re-coding it in S-Net, which delivered some useful insights, and to Dr A.Kudryavtsev of the Institute of Theoretical and Applied Mechanics, Russia, for writing part of the Particles-in-Cell simulation code, which aided our understanding of what was involved in using S-Net in the SPMD style when programming irregular CFD applications.

References

HPC Interconnection Networks: The Key to Exascale Computing

Jeffrey S. VETTER, Vinod TIPPARAJU, Weikuan YU and Philip C. ROTH
Future Technologies Group, Oak Ridge National Laboratory

Abstract. Scientists from many domains desire to address problems within the next decade that, by all estimates, require computer systems that can achieve sustained exaflop computing rates (i.e., $1 \times 10^{18}$ floating point operations per second) with real-world applications. Simply scaling existing designs is insufficient: analysis of current technological trends suggests that only a few architectural components are on track to reach the performance levels needed for exascale computing. The network connecting computer system nodes presents a particularly difficult challenge because of the prevalence of a wide variety of communication patterns and collective communication operations in algorithms used in scientific applications and their tendency to be the most significant limit to application scalability. Researchers at Oak Ridge National Laboratory and elsewhere are actively working to overcome these network-related scalability barriers using advanced hardware and software design, alternative network topologies, and performance prediction using modeling and simulation.

Keywords. Exascale, high-performance computing, interconnection networks.

Introduction

Computational scientists studying climate, nuclear physics, and fuel combustion share a need for extremely powerful computing resources in order to reach the desired problem scale or resolution. In 2008, the first High Performance Computing (HPC) systems capable of computing at a rate of one petaflop per second ($1 \times 10^{15}$ floating point operations per second) for general scientific workloads were deployed at Oak Ridge National Laboratory and Los Alamos National Laboratory in the United States. However, even these petascale systems are insufficient for some of the problems that computational scientists wish to address. Consequently, researchers and system designers are already considering how to implement exascale systems (those capable of $1 \times 10^{18}$ floating point operations per second). Extrapolating from historical performance data from the Top 500 list [1], at the current pace of technological progress exascale systems are expected to become available by approximately 2022. With enough investment that schedule may be accelerated to approximately 2017 [2].
Designing a system that is theoretically capable of exaflop computing is relatively easy. Although the trend of increasing clock speeds for commodity processors has stalled due to power and thermal constraints, the number of cores per processor is expected to continue to increase as processor architects make use of the increasing number of available transistors provided by an industry determined to hold to Moore’s Law. To date, increases in the number of cores per processor have implied an increase in the aggregate theoretical floating point capability of the processor with most commodity designs. To augment the computing capabilities of commodity processors, system architects are already incorporating compute accelerator devices such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Single Instruction Multiple Data (SIMD) accelerators into traditional system designs. The petascale Roadrunner system deployed at Los Alamos National Laboratory exhibits another type of architectural heterogeneity for increased computational rate, combining commodity x86 processors with more specialized, higher performance Cell processors in each compute node.

Whereas designing system hardware that is theoretically capable of computing at exaflop rates is expected to be relatively easy, achieving sustained exaflop rates with real-world scientific applications is expected to be much more difficult. Software technology such as compilers and runtime libraries can limit the scalability of real programs by making ineffective use of the available processor resources, and this problem is compounded by the trend toward increasing the number of processor cores without corresponding increases in the bandwidth between the processor, memory, and the interconnection network. Furthermore, real-world scientific applications very often use collective communication operations to transfer data between program processes, and those operations are often found to be the most significant barrier to program scalability. Thus, the true key to exascale computing is to design systems with high performance interconnection networks and to make effective use of those network resources.

In this chapter, we present the state of the art in interconnection network technology, including case studies of two important interconnection technologies: the open standard InfiniBand [3, 4] and the proprietary interconnection network used in the Cray XT platform. Then, we discuss the barriers to system scalability that must be addressed to make exascale computing feasible. Finally, we provide a brief overview of research that we and others are doing to address these scalability barriers.

1. State of the Art

A parallel computing system’s interconnection network consists of both hardware and software that allows processes running on distinct computing nodes within the system to transfer data to each other. At a minimum, network hardware includes a network interface card (NIC, sometimes called an adapter) that transfers data between the node’s memory and the network, and cables across which data is transferred between nodes. Because extreme-scale computers contain tens or even hundreds of thousands of nodes, a node in such a system cannot have a direct network connection to every other system node. Consequently, some network organizations use switches to multiplex and demultiplex connections between system nodes.
In this section we describe the current state of the art in interconnection network hardware and software. We also present case studies of two modern interconnection networks: the InfiniBand Architecture and the Cray XT interconnection network.

1.1. Hardware

An interconnection network can have a varying number of building blocks, including routers, switches, network interfaces, repeaters, connectors, and cables. Among them, a network interface connects a compute node to the network, serving the purposes of sending and/or receiving data. It is typically configured as a peripheral device connected to the node’s I/O bus, and is therefore called a network interface card (NIC). On an InfiniBand network, it is also referred to as channel adaptor (CA). Some network topologies use switches to enable connectivity between network interface cards in system nodes. Mesh- or torus-based interconnection networks avoid the need for such intermediate switches. On these networks, a network interface also serves as a router, directly forwarding the packets to other network interfaces using routing tables or routing algorithms.

1.1.1. Network Interfaces

Network interfaces are the source and destination for data sent across the interconnection network. Although details vary depending on the interconnection network, there are several features present in all modern network interface cards. To interact with both the compute node and the network, a network interface card contains (a) a host interface to receive commands, (b) a Direct Memory Access (DMA) engine to download or upload data across the I/O bus, (c) a processing unit, (d) network ports, and (e) physical input/output queues for networking packets on each port.

To avoid contention between application computation and processing of network data packets on the node’s processors, some modern network interface cards have the capability to offload some network processing tasks to the NIC. For instance, packet checksum generation and checking is often offloaded to the NIC. Also for performance reasons, modern high speed NICs also provide memory registration to pin down the volatile memory for network access allowing network transfers to occur without involving the node’s operating system kernel (an approach called OS bypass). This memory registration also serves as a mechanism to provide protection between different processes. Some networks (e.g. Quadrics) even provide an integrated memory management unit (MMU), along with the associated cache and Translation Look-aside Buffer (TLB) similar to that used by the node’s CPU(s). The NIC’s MMU page tables are kept consistent with the equivalent tables in the host virtual memory. Virtual addresses contained in networking packets can be translated into either local SDRAM physical addresses or bus physical addresses without the need for user processes to explicitly pin their memory. Although this approach is technically appealing and user friendly, its use is gradually fading because of its complexity and its propriety nature.

One of the standard data transfer mechanisms provided by nearly all modern high speed interconnects is Remote Direct Memory Access (RDMA). RDMA is a networking mechanism that enables one process to directly access the memory from another process on a remote node, without placing a heavy burden on either the memory bus or the host CPU on the remote node. RDMA enables zero-copy OS bypass networking: data is transferred directly to or from application memory without
applications issuing a system call from user space to the OS in order to access the NIC and copy the data to kernel space. This greatly reduces the number of costly traversals of the user-kernel boundary compared to traditional network protocol implementations.

Vendors are beginning to provide highly programmable processors in their network interface cards, providing a great deal of flexibility for the design and implementation of communication protocols. Coupled with OS bypass approaches, these programmable NICs provide a high-performance, flexible mechanism to optimize data inter-process communication between application processes on extreme-scale systems.

1.1.2. Switches

Switches are the backbone of a multi-stage interconnection network. They are responsible for forwarding network packets from a source node’s NIC to another at the destination. A switch provides some number of input/output ports. Leaf switches directly connected to system nodes may contain a small number of ports (e.g., a 36-port InfiniBand leaf switch) but switches that connect leaf switches together may provide several hundred ports to enable the construction of systems with a very large number of compute nodes without adding too many levels of switches. The various interconnection network types support a variety of switch-based network topologies. For example, Myrinet uses the CLOS topology whereas InfiniBand and Quadrics typically use a fat-tree. Network topology is discussed more fully later in this section.

The fundamental task of collection of switches is routing, i.e., to determine the correct path through the network for all the network packets. Early computer networks used circuit switching and store-and-forward techniques, but modern high speed interconnects use cut-through (also known as worm-hole). With cut-through routing, all packets are divided into flow control units or flits. A switch determines the correct output port as soon as the head flit is available in the input buffer. This greatly shortens the switching latency compared to earlier routing techniques, and at the same time minimizes the memory needed for buffer space in a switch. As a result, all modern interconnects support layer-2 switching at the network level, compared to the layer-3 switching that has to be implemented at the transport level.

Routing policies differ significantly across interconnection network types. Some technologies determine the route a packet will take at the packet’s source (source-based routing), while others use table-based lookup of the destination in each switch to determine a packet’s path (destination-based routing). Also, some switches can change the route a packet will take to avoid network congestion or failed links; this capability is called adaptive routing and contrasts with deterministic routing. For performance and resiliency reasons, source-based adaptive routing is a common combination provided by modern interconnection networks for large-scale systems.

Communication patterns in scientific applications often involve collective communication operations that involve more than just a <source,destination> pair of processes. For example, in a broadcast collective operation a single process delivers the same data to several other processes in one operation. For performance reasons, support for collective operations is being added to modern interconnection network switches and NICs to avoid the use of slower, software-only collective communication implementations.
1.2. Topologies

An interconnection network’s organization is called its topology. Early parallel computer designs explored a wide variety of topologies including rings, stars, and hypercubes, but most interconnection networks today use a tree, fabric, or an n-dimensional mesh topology.

In a tree topology, compute nodes are located at the leaves of a tree organization and one or more levels of switches are used to connect the nodes. When data is sent from a source node to a destination node, the sequence of switches through which the data must pass is completely determined by the relative positions of the nodes. Because the connections toward the root of the tree are involved in more such sequences than those toward the leaves of the tree, connections toward the root of the tree have the potential to be network bottlenecks. Fat tree topologies alleviate this problem by using higher bandwidth connections toward the root of the tree than at the leaves. In an ideal fat tree, the bandwidth between any two nodes is the same regardless of their location in the system. However, because of the large number of nodes in an extreme-scale system, providing full bandwidth at the root of a tree topology is cost prohibitive, leading to network organizations that oversubscribe internal tree connections at ratios of two-to-one (or more) compared to the links connecting nodes to network leaf switches.

A fabric topology is similar to a tree in that it uses levels of switches through which traffic must pass when transferred between nodes. Nodes are attached to small switches called leaf or edge switches at the edges of the fabric. Switches internal to the fabric may be arranged as a mesh or some other organization that provides redundant paths between any pair of network endpoints.

N-dimensional mesh network topologies avoid the large internal switches and potential for bandwidth problems common to tree and fabric organizations by organizing compute nodes in an n-dimensional mesh. In modern system designs, a three-dimensional mesh is common. With this type of mesh, each compute node is connected to six other compute nodes—its neighbors in the mesh. To reduce the network diameter compared to a strict mesh topology, additional network connections are often used to make the network topology a torus in each dimension. Assuming a good mapping of problem data to compute nodes, a three-dimensional mesh topology can be a good match for 3D simulations that rely heavily on nearest neighbor communication.

1.3. Software

The approach used by a parallel program to express parallelism is often called its programming model. For example, the message passing programming model is the predominant model used by scientific applications today. In particular, many scientific applications limit themselves to the two-sided communication operations defined in the Message Passing Interface (MPI) version 1 standard [5] for reasons such as portability and ease of implementation. By defining the available communication operations, a programming model defines the “language” that a program can use for communication. Because that language can either hide or expose the functionality of a system’s interconnection network, the selection of programming model can greatly affect how well an application utilizes the network. For instance, an InfiniBand network is capable
of true one-sided communication operations but applications that use only MPI-1 cannot access this one-sided communication capability directly.

To address these limitations of MPI-1, alternative programming models such as Partitioned Global Address Space (PGAS) and Global Address Space (GAS) have been developed that make better use of the advanced communication capabilities provided by modern networks. Implementations of PGAS languages like Co-array Fortran [6, 7] and Unified Parallel C [8], GAS libraries like Global Arrays [9] and communication libraries such as GASnet [10] directly expose one-sided operations or use concepts that are efficiently implemented as thin layers on top of such functionality.

With modern compute node operating systems, application programs do not interact directly with compute node hardware like the network interface card. Instead, intermediate software components control the hardware and provide an abstraction of the actual hardware to application programs (or to higher-level libraries like an MPI implementation or Global Arrays). Ideally, these software components expose the capabilities of the network hardware and enable efficient use of that hardware for transferring data. Unfortunately, poor software design and low quality software implementations can significantly reduce the application level communication performance.

1.4. Case Study: InfiniBand

The InfiniBand Architecture [3, 4] is a high-performance I/O technology. In today’s InfiniBand (IB) systems, the technology is used as a communication fabric that connects system nodes to each other and to storage devices.

In a system with an IB network, each system node interfaces to the IB fabric using a host channel adapter (HCA). An HCA is connected via a cable to a target channel adapter (TCA) within an IB switch or a peripheral. Although fabrics using switches are the most common topologies for IB network deployments, some IB HCAs support direct HCA-to-HCA connections enabling mesh and torus topologies.

IB links are bi-directional, serial connections between IB adapters. When signaled at IB’s base clock rate, the bandwidth of each link is 2.5 Gb/s in each direction. However, IB also supports signaling at a multiple of the base clock rate. Currently, double data rate (DDR) and quad data rate (QDR) products are available giving twice and four times, respectively, the bandwidth of a single data rate (SDR) link. Also, IB links can be aggregated for higher bandwidth than a single link; support for 4x and 12x aggregated IB links are common. These links are connected to the IB switches for access to the other nodes in the system. IB Switches allow for connecting system nodes in many different topologies, but the most common topology used in IB-based systems is a fat tree. For larger systems an oversubscribed fat tree is generally used. In an oversubscribed fat tree there are a fewer links going to the root of the tree than towards the leaves.

IB transmits data in messages. IB supports traditional two-sided send and receive message operations. IB also supports Remote Data Memory Access (RDMA) operations whereby one node writes data to, or reads data from, a memory buffer in another node. An atomic memory update operation is also supported. Messages are packetized for transmission across an IB channel. The IB standard defines a collection of operations (called verbs) that must be provided in an IB programming interface but does not specify their syntax. The Open Fabrics Enterprise Distribution (OFED) [11]
software stack provides a commonly used syntax and high-level libraries for using an IB network.

1.5. Case Study: Cray XT

The Cray XT [12, 13] is a parallel computing platform featuring commodity processors and a custom interconnection network, and supporting massively parallel system deployments. For instance, the petascale Jaguar Cray XT system [14] deployed at Oak Ridge National Laboratory (ORNL) has over 45,000 AMD Opteron processors and an interconnection network bisection bandwidth of 532 TB/s.

Within each XT node, a processor is connected to the XT interconnection network via a HyperTransport link to a custom Application Specific Integrated Circuit (ASIC) called SeaStar. SeaStar is a routing and communications chip that provides six high-speed links to other nodes in a 3D mesh topology. In earlier XT generations, each network link had a peak bandwidth of 7.6 GB/s but the peak bandwidth of the XT5 links has been raised to 9.6 GB/s. In the XT, all message passing traffic is carried on the interconnection network as well as all I/O traffic to the service nodes providing access to the parallel file system.

The XT uses the Portals [15] data movement layer for flexible, low-overhead internode communication. Portals provides connectionless, reliable, in-order delivery of messages between processes. For high performance and to avoid unpredictable changes in the kernel’s memory footprint, Portals deliver data from a sending process’ user space to the receiving process’ user space without kernel buffering. Portals supports both one-sided and two-sided communication models. Although any program can use Portals to transfer data between processes, application developers usually use a higher level programming interface such as Cray’s MPI implementation for data transfer.

2. Research Directions

There are many interconnection network challenges that must be addressed for effective exascale computing. Within the Future Technologies Group at ORNL we are working to address many of these challenges. We present a brief overview of several of our research activities in this section.

2.1. Addressing common communication patterns

Communication patterns give mathematical and graphical representations of how different processes in an application transfer data and signals among themselves. They are often used to map different processes in an application to the hardware topology of a system. Scientific applications demonstrate common communication patterns by their use of nearest neighbor communication and collectives. Several studies have targeted at determining these communication patterns for different applications both by tracing the steps in the execution of an application and profiling the application using a profiling tool. Techniques to extrapolate and predict the communication patterns have also been proposed [16]. For applications to run at peta- and exascale, these common communication patterns need to be supported in network for both performance and scalability. Support for addressing common communication patterns is also motivated
by recent advances in parallel programming models and in particular by the asynchronous style of parallel programming using one-sided communication in GAS and PGAS languages. Support in hardware for common communication patterns allows for communication between processors without interrupts or software overheads. Research in supporting different kinds of communication patterns is being pursued, notably in the fields of collective and one-sided communication.

2.1.1. Collective communication support in the interconnection network

One common occurrence of regular communication patterns in applications is via their use of collective communication calls. Various collective communication operations like Broadcast, Multicast, Reduce and Barrier are used frequently in scientific applications. Over the past decade, researchers have attempted to support these operations in the interconnection network, some via programming a programmable network (such as Quadrics QSNetI and QSNetII networks) and some by providing additional hardware for specialized communication. Notable software implementations of these are the implementations of Broadcast, Multicast and Barrier on the Quadrics QSNet network [17] and the Myricoms’ Myrinet network [18]. The collective communication network on the Blue Gene/L and Blue Gene/P is an excellent example of specialized hardware support for doing common collective communication operations. Several researchers at ORNL and other research institutions, in collaboration with the industry, are working on designing interconnection networks of the future with direct hardware and software support for common collective communication used in scientific applications.

2.2. One-sided communication APIs

Most of the modern interconnects have support for Remote Direct Memory Access (RDMA) which provides a mechanism to integrate into the memory subsystem of the host and provide access to the memory on the host via the network. However, programming models that rely on one-sided communication have a more complex and richer set of interfaces that support data-types representative of the science addressed by the scientific applications. The underlying RDMA mechanisms today are not capable of handling the requirements of these programming models. Several researchers are investigating additions to network hardware and extensions to the prevalent RDMA API to support true one-sided communication. These include support for one-sided operations such as one-sided reduction (also referred to as Accumulate) in the network hardware via specialized additions to the hardware, and in the network software via enriching the Remote Direct Memory Access API that the network provides.

2.3. Optimized Communication for I/O

Scientific applications use I/O for obtaining input files, writing checkpoint files as defense against system unreliability, and to save program output. Exascale computing platforms needs I/O capability commensurate with their memory capacity and computing power. Parallel I/O techniques provide this support the underpinning support by integrating all the building components of the I/O subsystem including storage, file system, networking and parallel I/O libraries. However, parallel I/O
libraries such as MPI-IO [19] have so far failed to adequately adapt to changes in the hierarchical composition of processing units on the same chip (or across the nodes), the topological layout of interconnects, and the changing scalability trends of computation, communication and I/O. In particular, collective communication was recently uncovered as a key bottleneck to the scalability and efficiency of parallel I/O. This bottleneck is referred to as collective wall, which, if left unaddressed, would lead to a colossal scalability challenge for platforms that have hundreds of thousands of processors and beyond. In addition, while the interconnect bandwidth is typically shared by both message passing and I/O traffic, one of them, e.g. message passing, can potentially assert bursts of traffic or various optimizations with neither knowledge nor consideration on the other’s needs and requirements, therefore causing efficiency degradation on the overall resource utilization. Thus, future exascale computing requires further research and development to delve into the challenges caused by collective communication to achieve extreme-scale parallel I/O.

2.4. Performance prediction

When evaluating research ideas, implementing new software and especially hardware can be prohibitively expensive. To mitigate this expense, we desire some idea of the impact of our proposed software and hardware modifications before we pay the implementation cost. This impact has many aspects, including performance, cost, power demand, heat generation, and physical form factor. Although all aspects are important, performance is a primary consideration: a large performance benefit may offset disadvantages in other areas.

To understand the performance impact of our research ideas before we have a complete implementation of them, we are developing, implementing and evaluating a performance prediction approach that combines two established performance prediction techniques: modeling and simulation. In the Future Technologies Group at ORNL, Modeling Assertions (MA) [20] is the primary performance modeling approach. In the MA approach, we annotate program code constructs with symbolic expressions regarding the work done by the construct. For instance, the annotation for a loop nest could indicate the number of iterations we expect the loop to perform, expressed in terms of the relevant program input parameters (or variables derived from such parameters). With respect to an application’s communication demands, the symbolic expressions express characteristics such as the number of bytes transferred and the processes involved in the communication operation. [21] At run time, a MA runtime component verifies the symbolic code annotations and combines symbolic expressions into expressions that represent larger program units such as modules or the whole program.

Understanding a parallel program’s computation and communication demands is important for performance prediction, but insufficient: one also needs to understand how well a specific system will service those demands. For example, knowing that a given communication operation transfers 5MB is useful, but with this information alone we cannot predict how long such a transfer will take. However, if we know the performance characteristics of the NIC, routers and/or switches, and network cables over which this transfer will occur, we can predict the latency for the transfer.
For a particular system, its performance characteristics can be captured using analytical models and using simulation. Because of the challenges in developing accurate analytical models for complex systems like the exascale systems for which we want to generate performance predictions, in the Future Technologies Group we have adopted a simulation-based approach to complement Modeling Assertions. In our approach, MA is used to capture and express application computation and communication demands. These expressions of program demands are then used as a workload specification for a discrete event simulation of a particular target system. For example, Figure 1 shows a display from a prototype simulation of a small Cray XT showing its 3D torus network. In effect, MA produces a high-level description of program behavior, and the simulator emulates this behavior, taking into account contention for limited system resources such as network link bandwidth. Because the simulator assigns timestamps to program events during its simulation, the simulator provides performance predictions for the simulated application running on a particular target system.

2.5. Other Exascale Barriers

In addition to the interconnection network-related challenges that we are actively working to address, there are also problems being addressed by colleagues in industry and the research community. One significant challenge is that of network cabling. Copper cables are commonly used in today’s interconnection network deployments, but copper cabling suffers from many problems when used in extreme-scale systems. In the amounts necessary for an extreme-scale interconnection network, copper cabling is heavy and must be relatively short to avoid excessive signal degradation. Optical cabling has been proposed as a lighter, smaller alternative that allows for longer cables but current optical-electrical conversion technology has not provided the desired signaling rates needed for overcoming performance barriers at the exascale. Silicon
photonics, whereby silicon-based optics are used for communication within a chip and between boards for high bandwidth, is an active research direction for addressing the problems of optical-electrical conversion.

Power requirements are a fundamental concern for systems designers in all components of a large-scale system. For instance, one of the primary design goals of IBM’s Blue Gene platform [22] is massive parallelism and good performance with controlled power requirements. Unfortunately, high performance usually requires high power, so interconnection network researchers are investigating ways to use low power components and to manage networking components so that overall power requirements are controlled. For example, custom network interfaces incorporating advanced power management features may turn off networking components during compute-bound phases of program execution.

For many scientific applications, high network bandwidth is the most critical characteristic allowing them to achieve good performance for collective communication operations. However, there are applications for which low network latency is important. The latency of a data transfer can be broken into three primary components: time required to traverse the networking software stack in the source and destination nodes, time required to traverse the network interface hardware, and time required to traverse the cable connecting the nodes. Although cable latency has been decreasing over the recent past, physical limits imposed by the speed of light through copper are causing the rate of decrease to slow and is expected to asymptotically level off in the near future. Thus, decreases in network latency must come from more efficient network interface hardware and especially through more efficient networking software stacks. Increased use of application and OS bypass approaches enabled by RDMA-capable hardware is promising but places an increased burden on high-level communication libraries such as the MPI implementation. Improving the efficiency of such libraries’ use of RDMA (or using programming models that are naturally able to use RDMA functionality) is a critical part of addressing the software-related networking scalability barriers.

3. Summary

At the current pace of technological change, systems that are capable of exascale computing will become available in the early- to mid-2020s. Experts from across the U.S. Department of Energy research community believe it is possible to deploy exascale computers addressing real-world science problems in approximately 2017, but only by overcoming several significant barriers to application scalability. Chief among these are limits to collective communication performance. We, and others throughout the research community, are actively working to address these performance barriers using advanced hardware design including hardware support for collective communication, optimized collective communication patterns, and performance prediction using simulation and performance modeling.
Acknowledgements

This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. De-AC05-00OR22725.

References

Chapter 3
GRID Technologies
This page intentionally left blank
Using Peer-to-Peer Dynamic Querying in Grid Information Services

Domenico TALIA\textsuperscript{a,b} and Paolo TRUNFIO\textsuperscript{a,1}
\textsuperscript{a} DEIS - University of Calabria, Italy
\textsuperscript{b} ICAR-CNIR, Italy

Abstract. Dynamic Querying (DQ) is a technique adopted in unstructured Peer-to-Peer (P2P) networks to minimize the number of nodes that is necessary to visit to obtain the desired number of results. In this chapter we describe the use of the DQ technique over a Distributed Hash Table (DHT) to implement a scalable Grid information service. The DQ-DHT (Dynamic Querying over a Distributed Hash Table) algorithm has been designed to perform DQ-like searches over DHT-based networks. The aim of DQ-DHT is two-fold: allowing arbitrary queries to be performed in structured P2P networks, and providing dynamic adaptation of search according to the popularity of resources to be located. Through the use of the DQ-DHT technique it is possible to implement a scalable Grid information service supporting both structured search and execution of arbitrary queries for searching Grid resources on the basis of complex criteria or semantic features.

Keywords. Grid, Peer-to-Peer, Dynamic Querying, Distributed Hash Tables

Introduction

Grid applications often require a large number of distributed resources that need to be discovered and selected on the basis of user requirements and system constraints. The goal of a Grid information service is providing the basic mechanisms to index and discover all the resources (processors, memories, software, etc.) required to run complex Grid applications.

Designing an efficient Grid information service is a challenging task as classical architectures based on hierarchical models do not scale in large-scale Grid environments. To improve scalability, the Peer-to-Peer (P2P) approach has been proposed as an alternative to hierarchical models to implement Grid information services in such large-scale scenarios. Indeed, several P2P systems have been proposed so far to enable scalable resource discovery in Grids [1]. Those systems are classified either as \textit{unstructured} or \textit{structured}, according to the way nodes are linked to each other and information about resources is placed on nodes.

In unstructured systems (e.g., [2] and [3]) links among nodes can be established arbitrarily and data placement is unrelated to the topology of the resulting overlay. In such systems, when a node wishes to find a given resource, the query must be distributed...
through the network using flooding-like techniques to reach as many nodes as needed. Each node reached by the query processes it on the local data items and, in case of match, replies to the query initiator.

Structured systems, like MAAN [4] and XenoSearch [5], keep association of resource identifiers to nodes using a Distributed Hash Table (DHT), which allows to locate the node responsible for the resource with a given Id (or key) with logarithmic performance bounds. Structured systems, however, do not support arbitrary types of queries (e.g., regular expressions [6]) because it is infeasible to generate and store keys for every query expression. On the contrary, unstructured systems can do it effortlessly since all queries are processed locally on a node-by-node basis [7].

Even if the lookup mechanisms of DHT-based structured systems do not support arbitrary queries, it is possible to exploit their structure to distribute any kind of information across the overlay with minimal cost. For example, in [8] a technique for efficient broadcast over a DHT is proposed. Using such technique, a broadcast message originating at an arbitrary node in the DHT overlay reaches all other nodes without redundant messages in $O(\log N)$ steps. It can be used to broadcast arbitrary types of queries, which can then be processed locally by single nodes as in unstructured systems.

We followed this approach by designing a P2P search algorithm, named DQ-DHT (Dynamic Querying over a DHT), to provide efficient execution of arbitrary queries in structured P2P networks [9]. DQ-DHT is based on a combination of the broadcast technique mentioned above with the Dynamic Querying (DQ) technique [10] used in unstructured networks.

The goal of DQ is to minimize the number of nodes that is necessary to visit in an unstructured network to obtain the desired number of results. The query initiator starts the search by sending the query to a few of its neighbors and with a small Time-To-Live (TTL). The main goal of this “probe” query is to estimate the popularity of the resource to be located. If such an attempt does not produce a sufficient number of results, the search initiator sends the query towards the next neighbor with a new TTL. Such TTL is calculated taking into account both the desired number of results and the resource popularity estimated during the previous phase. This process is repeated until the expected number of results is received, or there are no more neighbors to query.

Similarly to DQ, DQ-DHT performs the broadcast in an iterative way until the target number of results is obtained. At each iteration, a new subset of nodes is queried on the basis of the estimated resource popularity and the desired number of results. In contrast to DQ, DQ-DHT exploits the structural constraints of the DHT to avoid message duplications and ensure higher success rate.

DQ-DHT has been particularly designed to support arbitrary queries over existing DHT-based Grid information services. Hence, our approach is to use the DHT overlay for two purposes: 1) indexing Grid resources based on attribute-value pairs using standard DHT techniques to support structured search, including multi-attribute [4], keyword-based [11], and range queries [12]; 2) distributing arbitrary queries across nodes for subsequent local processing using the DQ-DHT algorithm, in order to support unstructured search of Grid resources based on complex criteria or semantic features that cannot be expressed as simple combination of attribute-value pairs.

In this chapter we describe the DQ-DHT algorithm using Chord [13] as the DHT overlay. We also describe an extension of DQ-DHT allowing to perform dynamic querying search in a $k$-ary DHT-based overlay [14]. In a $k$-ary DHT, broadcast takes only
$O(\log_k N)$ hops using $O(\log_k N)$ pointers per node. We exploited this “$k$-ary principle” in DQ-DHT to improve the search time with respect to the Chord-based implementation.

The rest of the chapter is organized as follows. Section 1 provides a background on the technique of broadcast over a DHT exploited by DQ-DHT. Section 2 describes the DQ-DHT algorithm. Section 3 describes how DQ-DHT is implemented over a $k$-ary DHT. Section 4 presents a performance evaluation of DQ-DHT over Chord and over a $k$-ary DHT in different scenarios. Section 5 discusses related work. Finally, Section 6 concludes the chapter.

1. Broadcast over a DHT

This section briefly describes the Chord-based implementation of the broadcast algorithm proposed in [8].

Chord uses a consistent hash function to assign each node an $m$-bit identifier, which represents its position in a circular identifier space ranging from 0 and $2^m - 1$. Each node, $x$, maintains a finger table with $m$ entries. The $j^{th}$ entry in the finger table at node $x$ contains the identity of the first node, $s$, that succeeds $x$ by at least $2^j - 1$ positions on the identifier circle, where $1 \leq j \leq m$. Node $s$ is called the $j^{th}$ finger of node $x$. If the identifier space is not fully populated (i.e., the number of nodes, $N$, is lower than $2^m$), the finger table contains redundant fingers. In a network of $N$ nodes, the number $u$ of unique (i.e., distinct) fingers of a generic node $x$ is likely to be $\log_2 N$ [13]. In the following, we will use the notation $F_i$ to indicate the $i^{th}$ unique finger of node $x$, where $1 \leq i \leq u$.

To perform the broadcast of a data item $D$, a node $x$ sends a BROADCAST message to all its unique fingers. The BROADCAST message contains $D$ and a limit argument, which is used to restrict the forwarding space of a receiving node. The limit sent to $F_i$ is set to $F_{i+1}$, for $1 \leq i \leq u - 1$. The limit sent to the last unique finger, $F_u$, is set to the identifier of the sender, $x$. When a node $y$ receives a BROADCAST message with a data item $D$ and a given limit, it is responsible for forwarding $D$ to all its unique fingers in the interval $[y, \text{limit}]$. When forwarding the message to $F_i$, for $1 \leq i \leq u - 1$, $y$ supplies it a new limit, which is set to $F_{i+1}$ if it does not exceed the old limit, to the old limit otherwise. As before, the new limit sent to $F_u$ is set to $y$. As shown in [8], in a network of $N$ nodes, a broadcast message originating at an arbitrary node reaches all other nodes after exactly $N - 1$ messages, with $O(\log_2 N)$ steps.

Figure 1a shows an example of broadcast in a fully populated Chord ring, where $u = m = 4$. For each node, the corresponding finger table is represented. The BROADCAST messages are represented by rectangles containing the data item $D$ and the limit parameter. The entire broadcast is completed in $u = 4$ steps, represented with solid, dashed, dashed-dotted, and dotted lines, respectively.

In this example, the broadcast is initiated by Node 2, which sends a BROADCAST message to all nodes in its finger table (Nodes 3, 4, 6 and 10) (step 1). Nodes 3, 4, 6 and 10 in turn forward the BROADCAST message to their fingers under the received limit value (step 2). The same procedure applies iteratively, until all nodes in the network are reached (steps 3 and 4).

The overall broadcast procedure can be viewed as the process of passing the data item through a spanning tree that covers all nodes in the network [8]. Figure 1b shows the
spanning tree corresponding to the example of broadcast shown in Figure 1a. The root of the spanning tree is the node that initiates the broadcast (Node 2). The tree is composed of four subtrees, each one having, as root, one of the fingers of Node 2 (that is, Nodes 3, 4, 6 and 10). Since the spanning tree corresponds to the lookup tree, which is a binomial tree in a (fully populated) Chord network [15], also the spanning tree associated to the broadcast over a fully populated Chord ring is a binomial tree.

2. Dynamic Querying over a DHT

The DQ-DHT algorithm works as follows. Let \( x \) be the node that initiates the search, \( U \) the set of unique fingers not yet visited, and \( R_d \) the desired number of results. Initially \( U \) includes all unique fingers of \( x \). Node \( x \) starts by choosing a subset \( V \) of \( U \) and sending the query to all fingers in \( V \). These fingers will in turn forward the query to all nodes in the portions of the spanning tree they are responsible for, following the broadcast algorithm described above. When a node receives a query, it checks for local items matching the query criteria and, for each matching item, sends a query hit directly to \( x \). The fingers in \( V \) are removed from \( U \) to indicate that they have been already visited.

After sending the query to all nodes in \( V \), \( x \) waits for an amount of time \( T_L \), which is the estimated time needed by the query to reach all nodes, up to a given level \( L \), of the subtrees rooted at the unique fingers in \( V \), plus the time needed to receive a query hit from those nodes. Then, if the current number of received query hits \( R_c \) is equal or greater than \( R_d \), \( x \) terminates. Otherwise, an iterative procedure takes place.

At each iteration, node \( x \):

1. Calculates the item popularity \( P \) as the ratio between \( R_c \) and the number of nodes already theoretically queried;
2. Calculates the number $H_q$ of hosts in the network that should be queried to hit $R_d$ query hits based on $P$;

3. Chooses, among the nodes in $U$, a new subset $V'$ of unique fingers whose associated subtrees cumulatively contain the minimum number of nodes that is greater than or equal to $H_q$;

4. Sends the query to all nodes in $V'$;

5. Waits for an amount of time needed to propagate the query to all nodes in the subtrees associated to $V'$.

The iterative procedure above is repeated until the desired number of query hits is reached, or there are no more fingers to contact. Note that, if the item popularity is properly estimated after the first phase of search, only one additional iteration may be sufficient to obtain the desired number of query hits.

An important point in DQ-DHT is estimating the number of nodes present in the different subtrees, and at different levels, of the spanning tree associated to the broadcast process. In the next section we discuss how we calculate such properties of the spanning tree and introduce some functions that are used in the algorithm (described in Section 2.2).

2.1. Properties of the Spanning Tree Associated to the Broadcast over a Chord DHT

As recalled in Section 1, the spanning tree associated to the broadcast over a fully populated Chord ring is a binomial tree. A binomial tree of order $i \geq 0$, $B_i$, consists of a root with $i$ subtrees, where the $j^{th}$ subtree is a binomial tree of order $j - 1$, with $1 \leq j \leq i$. Given a binomial tree $B_i$, the following properties can be proven [16]: 1) The number of nodes in $B_i$ is $2^i$; 2) The depth of $B_i$ is $i$; 3) The number of nodes at level $l$ in $B_i$ is given by the binomial coefficient $\left(\binom{i}{l}\right)$.

Given the binomial tree properties, we can calculate the properties of the spanning tree associated to a broadcast initiated by a node having $u$ unique fingers (see Table 1).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_i$</td>
<td>Number of nodes in the subtree rooted at $F_i$, where $1 \leq i \leq u$</td>
<td>$2^{i-1} \times c$</td>
</tr>
<tr>
<td>$D_i$</td>
<td>Depth of the subtree rooted at $F_i$, where $1 \leq i \leq u$</td>
<td>$\log_2 N_i$</td>
</tr>
<tr>
<td>$N^l_i$</td>
<td>Number of nodes at level $l$ of the subtree rooted at $F_i$, where $1 \leq i \leq u$ and $0 \leq l \leq D_i$</td>
<td>$\left(\binom{D_i}{l}\right)$</td>
</tr>
</tbody>
</table>

Basically, in Table 1 we correct the binomial tree properties by a factor $c = N/2^u$, where $N$ is the number of nodes in the network (which can be estimated [17]), to compensate the fact that the value of $u$ may be different from the value of $\log_2 N$ in case of not fully populated rings. Note that, since the value of $D_i$ may be not an integer, we use the generalized binomial coefficient to calculate $N^l_i$.

Based on the spanning tree properties defined in Table 1, we define in Table 2 some aggregate functions operating on a set of unique fingers. Such functions are used in the DQ-DHT algorithm presented in the next section.
DQ-DHT defines two procedures: SUBMIT QUERY, executed by a node to submit a query, and PROCESS QUERY, executed by a node receiving a query to process.

### SUBMIT QUERY Procedure

<table>
<thead>
<tr>
<th>Function</th>
<th>Returned result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N(V)$</td>
<td>Total number of nodes in the subtrees associated to the unique fingers in $V$</td>
<td>$\sum_{i=1 \ldots i_n} N_i$</td>
</tr>
<tr>
<td>$D(V)$</td>
<td>Depth of the subtree associated to the unique finger with highest index in $V$</td>
<td>$D_i$ where $i = \max(i_1 \ldots i_n)$</td>
</tr>
<tr>
<td>$N(V, L)$</td>
<td>Total number of nodes from level 0 to level $L$ of the subtrees associated to the unique fingers in $V$</td>
<td>$\sum_{i=1 \ldots i_n} \sum_{i=0}^{l_i} N_i^l$ where $l_i = \min(L, D_i)$</td>
</tr>
</tbody>
</table>

2.2. **DQ-DHT Algorithm**

SUBMIT QUERY (see Figure 2) receives the query $Q$ and the desired number of results $R_d$. It makes use of the functions defined in Table 2, and it is assumed that the procedure is executed by a node $x$.

The procedure starts by initializing to 0 the current number of results $R_c$ (line 1). The value of $R_c$ is incremented by 1 whenever a query hit is received. A set $U$ is initialized to

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$R_c \leftarrow 0$</td>
</tr>
<tr>
<td>2</td>
<td>$U \leftarrow$ all unique fingers of node $x$</td>
</tr>
<tr>
<td>3</td>
<td>$H_v \leftarrow N(U)$</td>
</tr>
<tr>
<td>4</td>
<td>$V \leftarrow$ a subset of $U$</td>
</tr>
<tr>
<td>5</td>
<td>$U \leftarrow U \setminus V$</td>
</tr>
<tr>
<td>6</td>
<td>$L \leftarrow$ an integer $\in [0, D(V)]$</td>
</tr>
<tr>
<td>7</td>
<td>$T_L = T_H \times (L + 2)$</td>
</tr>
<tr>
<td>8</td>
<td>Send($Q, V$)</td>
</tr>
<tr>
<td>9</td>
<td>sleep($T_L$)</td>
</tr>
<tr>
<td>10</td>
<td>$H_v \leftarrow N(V, L)$</td>
</tr>
<tr>
<td>11</td>
<td>$T_r = T_H \times (D(V) - L)$</td>
</tr>
<tr>
<td>12</td>
<td>while $R_c &lt; R_d$ and $U \neq \emptyset$ do</td>
</tr>
<tr>
<td>13</td>
<td>if $R_c &gt; 0$ then</td>
</tr>
<tr>
<td>14</td>
<td>$P \leftarrow R_c / H_v$</td>
</tr>
<tr>
<td>15</td>
<td>$H_d \leftarrow R_d / P$</td>
</tr>
<tr>
<td>16</td>
<td>else</td>
</tr>
<tr>
<td>17</td>
<td>$H_d \leftarrow H_d + 1$</td>
</tr>
<tr>
<td>18</td>
<td>end if</td>
</tr>
<tr>
<td>19</td>
<td>if $H_d \leq N(V)$ then</td>
</tr>
<tr>
<td>20</td>
<td>sleep($T_r$)</td>
</tr>
<tr>
<td>21</td>
<td>$H_v \leftarrow N(V)$</td>
</tr>
<tr>
<td>22</td>
<td>$T_r \leftarrow 0$</td>
</tr>
<tr>
<td>23</td>
<td>else</td>
</tr>
<tr>
<td>24</td>
<td>$H_q \leftarrow H_d - N(V)$</td>
</tr>
<tr>
<td>25</td>
<td>if $H_q &gt; N(U)$ then</td>
</tr>
<tr>
<td>26</td>
<td>$V' \leftarrow U$</td>
</tr>
<tr>
<td>27</td>
<td>else</td>
</tr>
<tr>
<td>28</td>
<td>$V' \leftarrow$ subset of $U$ with min. $N(V') \geq H_q$</td>
</tr>
<tr>
<td>29</td>
<td>end if</td>
</tr>
<tr>
<td>30</td>
<td>$U \leftarrow U \setminus V'$</td>
</tr>
<tr>
<td>31</td>
<td>$T_{V'} = T_H \times (D(V') + 2)$</td>
</tr>
<tr>
<td>32</td>
<td>Send($Q, V'$)</td>
</tr>
<tr>
<td>33</td>
<td>sleep(max($T_{V'}, T_r$))</td>
</tr>
<tr>
<td>34</td>
<td>$H_v \leftarrow N(V) + N(V')$</td>
</tr>
<tr>
<td>35</td>
<td>$V \leftarrow V'$</td>
</tr>
<tr>
<td>36</td>
<td>$T_r \leftarrow 0$</td>
</tr>
<tr>
<td>37</td>
<td>end if</td>
</tr>
<tr>
<td>38</td>
<td>end while</td>
</tr>
</tbody>
</table>

- **Figure 2.** The SUBMIT QUERY procedure.
contain all unique fingers of node $x$ (line 2), and $H_s$ is set to $N(U)$, which corresponds to the total number of hosts that can be queried in the network (line 3). The first subset $V$ of fingers to visit is selected from $U$ (line 4), and $U$ is updated accordingly (line 5).

Afterwards, an integer $L$ between 0 and $D(V)$ is chosen (line 6). The value of $L$ represents the last level of the subtrees associated to $V$ from which to wait a response before to estimate the item popularity. The amount of time $T_L$ needed to receive a response from those levels is then calculated as $T_H \times (L + 2)$, where $T_H$ is the average time to pass a message from node to node (line 7). The value $L + 2$ is obtained by counting one hop to pass the message from $x$ to the fingers, $L$ hops to propagate the message up to level $L$, and an additional hop to return the query hit to node $x$.

Then, $Q$ is sent to all fingers in $V$ invoking the subroutine SEND described below (line 8). After the wait (line 9), the number of nodes visited $H_v$ is initialized to $N(V, L)$ (line 10). While the popularity will be estimated considering only levels from 0 to $L$, the query continues to be forwarded up to level $D(V)$. The additional amount of time $T_r$ that would be necessary to get a response from the remaining levels is therefore proportional to $D(V) - L$ (line 11).

After this first phase, an iterative process takes place while $R_c < R_d$ and there are more fingers to visit ($U \neq \emptyset$) (line 12). If at least one result has been received, node $x$ estimates the item popularity $P$ (line 14), and the estimated number $H_d$ of hosts to obtain $R_d$ results based on $P$ (line 15). Otherwise (i.e., $R_c = 0$), $H_d$ is set to $H_s + 1$, meaning that it is likely that more than all available hosts must be contacted to hit $R_d$ results (line 17).

If $H_d < N(V)$, it is expected to receive enough results from the fingers that have been already contacted. Note that this may happen only if $L < D(V)$, because $P$ is estimated on the basis of the results arriving from nodes up to level $L$ of the subtrees associated to $V$. Thus, only in this case, the search initiator must wait for the additional amount of time $T_r$ (line 20). After the wait, the value of $H_v$ is updated to include all nodes in $V$ (line 21), and $T_r$ is set to 0 (line 22).

Otherwise ($H_d > N(V)$), the number of nodes to be queried $H_q$ is given by $H_d$ minus the number of nodes already queried (line 24). If $H_q$ is greater than the number of nodes available, the new set $V'$ of fingers to visit is set to $U$ (line 26). Else, $V'$ is the subset of $U$ with the minimum value of $N(V')$ which is greater than or equal to $H_q$ (line 28). The elements in $V'$ are removed from $U$ (line 30), and the time $T_{V'}$ needed to receive response from all levels of the subtrees associated to $V'$ is calculated (line 31).

After sending the query to all nodes in $V'$ (line 32), $x$ performs a wait (line 33), updates the number of hosts visited (line 34), and sets $V$ to $V'$ (line 35). The waiting time on line 33 is the maximum between $T_{V'}$ and $T_r$, for managing the case in which the time $T_r$ needed to visit the levels remaining from the previous phase is greater than the time $T_{V'}$ needed to receive a response from all levels in $V'$. As for lines 19-22, this may happen only on the first iteration, since after that the timeout is always set to be proportional to $D(V')$, and so $T_r = 0$ (line 36).

The subroutine SEND forwards the query $Q$ to a set of unique fingers $V$. Basically, it implements the procedure executed by a node $x$ to perform a broadcast (see Section 1). The only difference is that we do not send the message to all unique fingers of $x$, but only to those in $V$. The message $M$ sent by $x$ to a node $y$ includes the Id of the querying node ($x$), the query to be processed $Q$, and the limit parameter used to restrict the forwarding space of node $y$. 


PROCESSQUERY (see Figure 3) is executed by a node $y$ that receives a message $M$ containing the Id of the search initiator $x$, the query to process $Q$, and the limit parameter. The procedure broadcasts the query to all nodes in the portion of the spanning tree node $y$ is responsible for (lines 1-16), following the broadcast algorithm described in Section 1. Then, it processes the query against its local resources, and for each matching item sends a query hit directly to the search initiator (lines 17-19).

\begin{verbatim}
PROCESSQUERY($M = \{x, Q, \text{limit}\}$)

1: for $i = 1$ to $u$ do
2: if $F_i \in \{y, \text{limit}\}$ then
3: if $i < u$ then
4: oldLimit $\leftarrow$ limit
5: limit $\leftarrow F_{i+1}$
6: if limit $\notin \{y, \text{oldLimit}\}$ then
7: limit $\leftarrow$ oldLimit
8: end if
9: else
10: limit $\leftarrow y$
11: end if
12: send message $M = \{x, Q, \text{limit}\}$ to $F_i$
13: else
14: exit for
15: end if
16: end for
17: for each local item matching $Q$ do
18: send query hit to node $x$
19: end for

Figure 3. The PROCESSQUERY procedure.
\end{verbatim}

3. Dynamic Querying over a $k$-ary DHT

In a $k$-ary DHT, pointers are placed to achieve a time complexity of $O(\log_k N)$, where $N$ is the number of nodes in the network and $k$ is some predefined constant. This is referred to as doing $k$-ary lookup or placing pointers according to the “$k$-ary principle” [18].

Let $M = k^m$ be the size of the identifier space, for some positive integer $m$. To achieve $k$-ary lookup, each node $x$ keeps $n_p = (k - 1) \times m$ pointers (or fingers) in its finger table. Each of these fingers can be chosen to be the first node that succeeds the start of every interval $f(j)$, where $f(j) = (x + c) \mod M$, and $c = (1 + ((j - 1) \mod (k - 1))) \times k^{\frac{m-1}{k-1}}$, for $1 \leq j \leq n_p$. For $k = 2$, the intervals coincide with those of Chord. If the identifier space is not fully populated (i.e., $N < M$), the finger table contains redundant fingers. In a network of $N$ nodes, the number $u$ of unique fingers of a generic node $x$ is likely to be $(k - 1) \times \log_k N$.

The broadcast algorithm described in Section 1, which is exploited by DQ-DHT as described in Section 2.2, can also be used in a $k$-ary DHT. In such case, the whole broadcast process takes only $O(\log_k N)$ hops. This can be illustrated as in Section 1 using a spanning tree view to represent the broadcast process over a $k$-ary DHT. As an example, Figure 4 shows the spanning tree corresponding to the broadcast initiated by Node 0 in a fully populated $k$-ary DHT with $k = 4$ and $N = 64$, while Figure 5 shows the spanning tree in a fully populated Chord network with the same size.

By comparing Figure 4 with Figure 5, it can be noted that the number of hops (that is, the depth of the spanning tree) needed to complete the broadcast in a $k$-ary DHT with $N = 64$ nodes passes from 5 with $k = 2$ (i.e., with Chord), to 3 with $k = 4$. We exploit this principle by extending DQ-DHT to improve the search time with respect to the original Chord-based implementation.
Figure 4. Spanning trees corresponding to the broadcast initiated by Node 0 in a fully populated \(k\)-ary DHT with \(k = 4\) and \(N = 64\).

Figure 5. Spanning trees corresponding to the broadcast initiated by Node 0 in a fully populated Chord network with \(N = 64\).

3.1. Properties of the Spanning Tree Associated to the Broadcast over a \(k\)-ary DHT

Since for \(k \neq 2\) the spanning tree is no more a binomial tree, we experimentally generalized the formulas presented in Table 1 to be applicable to the broadcast over a \(k\)-ary DHT, for any fixed \(k\). Table 3, in particular, shows how we calculate the properties of the spanning tree associated to the broadcast process in case of fully populated identifier space.

Table 3. Properties of the spanning tree in a fully populated \(k\)-ary DHT (see Table 1 for the definition of properties).

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N_i)</td>
<td>(N/(k^{(\left\lfloor \frac{u-1}{k-1} \right\rfloor )}+1))</td>
</tr>
<tr>
<td>(D_i)</td>
<td>(\log_k N_i)</td>
</tr>
<tr>
<td>(N_i^l)</td>
<td>((D_i^l) \times (k - 1)^l)</td>
</tr>
</tbody>
</table>

To verify the validity of the formulas in case of not fully populated identifier spaces, we employed a network simulator (the same used for the performance evaluation presented in Section 4). Through the simulator we built several random \(k\)-ary DHT overlays with different values of \(k\), and compared the real properties of the broadcast spanning tree with the values computed using the formulas in Table 3. The results of such experiments are summarized in Figure 6.
Figure 6. Comparison between computed and real values of $N_i$ and $N^l_i$ for different values of $k$, $i$ and $l$, in a simulated $k$-ary DHT with $N = 20000$ and $m = 20$. Lines represent the computed values. Single points with error bars represent the real values. The error bars of the real values represent the standard deviations from the mean, obtained from 100 simulation runs. All values of $N_i$ and $N^l_i$ are computed or measured from nodes with the following values of $u$: 15 for networks with $k = 2$; 18 for $k = 3$; 24 for $k = 5$; 32 for $k = 8$.

Figure 6a compares computed and real (i.e., measured) values of $N_i$ for different values of $i$, in a $k$-ary DHT with 20000 nodes and $m = 20$, considering the following values of $k$: 2, 3, 5, and 8. As shown by the graph, the means of the real values (represented as points) are very close to the computed values (represented as lines) for any value of $i$ and $k$.

The graph in Figure 6b considers again a $k$-ary DHT with $N = 20000$ and $m = 20$, but with $k$ fixed to 3, and compares computed and real values of $N^l_i$ for different values of $i$, with $l$ ranging from 1 to 4. As before, the mean of the real values resulted very close to the computed values for any value of $i$ and $l$.

In summary, the experimental results demonstrate that the formulas in Table 3 can also be used to estimate - with high accuracy - the properties of the spanning tree associated to the broadcast process in not fully populated $k$-ary DHTs.

3.2. Minor Modifications to the Original DQ-DHT Algorithm

The original DQ-DHT algorithm described in Section 2.2 works correctly over a $k$-ary DHT using the formulas defined in Table 3. In particular: i) the $N^l_i$ formula is used during the probe query to calculate the number of nodes theoretically queried after a predefined amount of time (which corresponds to the number of nodes up to a given depth in the subtrees rooted at the fingers queried during the probe phase); ii) the $N_i$ formula is used both to calculate the number of nodes already theoretically queried (given the set of unique fingers already contacted), and to choose a new subset of unique fingers to contact based on the theoretical number of nodes to query.

Even if the original DQ-DHT algorithm works properly for any value of $k$, we slightly modified it to obtain a more uniform comparison of its performance when different values of $k$ are used. The difference between the original version and the new one is explained in the following.

As discussed in Section 2.2, to perform the probe query the original algorithm needs two parameters: 1) the initial value of $V$, which is the first subset of unique fingers to
which the query has to be sent to; and 2) $L$, the last level of the subtrees associated to $V$
from which to wait a response before to estimate the resource popularity.

In the $k$-ary version, we replaced the two parameters above with the following: 1) $H_P$, defined as the number of hosts that will receive the query as a result of the probe phase; 2) $H_E$, the number of hosts to query before estimating the resource popularity.

Given $H_P$ and the set $U$ of unique fingers of the querying node, the algorithm calculates the initial set $V$ of unique fingers to contact as the subset of $U$ whose associated subtrees have the minimum number of nodes greater than or equal to $H_P$. In other terms, in the original algorithm the fingers to contact during the probe query are chosen explicitly, whereas in the $k$-ary version they are selected automatically based on the value of $H_P$.

While $H_P$ indicates the total number of nodes in the subtrees that will be flooded as a result of the probe phase, $H_E$ is the minimum number of nodes that must have received the query before estimating the resource popularity ($H_E \leq H_P$). Given $H_E$ and the initial set $V$ (calculated through $H_P$), the algorithm calculates the minimum number $L$ of levels of the subtrees associated to $V$ that contain a number of nodes greater than or equal to $H_E$. Therefore, $H_E$ is used in the $k$-ary version as an indirect way of specifying the value of $L$.

Since $H_P$ and $H_E$ are independent from the actual number of unique fingers and from the depth of the corresponding subtrees, their use allows to compare the algorithm performance using different values of $k$, independently from the number of pointers per node they produce in the resulting overlay.

4. Performance Evaluation

We evaluated the behavior of DQ-DHT over Chord and over a $k$-ary DHT using a custom-made discrete-event simulator written in Java. Two performance parameters have been evaluated: number of messages ($N_m$) and search time ($T_s$). $N_m$ is the total number of messages generated during the search process, while $T_s$ is the amount of time needed to receive the desired number of results. Our goal is understanding which are the algorithm parameters to use based on application requirements and system objectives (i.e., minimizing the number of messages or the search time).

The algorithm parameters of the Chord-based version are: the initial set of unique fingers to visit ($V$), the initial number of levels ($L$), and the desired number of results ($R_d$). For the $k$-ary version the algorithm parameters are: $H_P$ and $H_E$ (defined in Section 3.2) and the desired number of results, $R_d$.

The system parameters are: the number of nodes in the network ($N$) and the resource replication rate ($r$), where $r$ is the ratio between the total number of resources satisfying the query criteria and $N$. For both versions, all tests have been performed in a network with $N = 50000$ nodes and a value of $r$ ranging from 0.25 % to 32 %. The value of $N$ chosen corresponds to the largest network that our simulator was able to manage using our computing facilities.

Section 4.1 and Section 4.2 report the performance of DQ-DHT over Chord and over a $k$-ary DHT, respectively. All the results presented in both sections have been calculated as an average of 100 independent simulation runs, where at each run the search is initiated by a randomly chosen node.
4.1. DQ-DHT over Chord

We ran a first set of simulations to evaluate the behavior of DQ-DHT over Chord varying the initial set $V$ of unique fingers to contact. At each run we chose $V$ to include one of the fingers between $F_8$ to $F_{14}$, with the initial value of $L$ fixed to 5, and $R_d$ set to 100. Note that, even if it is possible to choose $V$ to include an arbitrary subset of the unique fingers of the querying node, we considered the case in which $V$ includes only one unique finger. This allows the algorithm to have, after the probe query, still $u - 1$ unique fingers from which to choose the new set of subtrees to query, this way improving the granularity of search.

The graphs in Figure 7 show number of messages and search time as a function of the replication rate. The search time is expressed in time units, where one time unit corresponds to the amount of time, $T_H$, needed to pass a message from node to node. Since in our simulations $T_H$ is fixed and equal to 1 for all nodes, the search times discussed below should be considered as an indication of the search times that could be obtained in a real network.

As expected, Figure 7a shows that the number of messages decreases as the replication rate increases, for any value of $V$. When $V = \{F_8\}$, the average number of messages passes from 48735 for $r = 0.25\%$, to 360 for $r = 32\%$. In the opposite case, $V = \{F_{14}\}$, the number of messages passes from 46473 for $r = 0.25\%$, to 8159 for $r = 32\%$.

For high values of $r$ (i.e., $r = 16 - 32\%$), in most cases the probe query is sufficient to obtain the desired number of results, and so the number of messages corresponds to the number of nodes in the subtree associated to the finger in $V$.

For values of $r$ lower than 2\%, typically at least one additional iteration after the probe query is needed. In these cases, the generated number of messages depends on the accuracy of the popularity estimation, which is better when a higher number of nodes is queried during the probe query (that is, when $V$ includes a finger with a high index). For instance, when $r = 1\%$, the average number of messages is 25207 for $V = \{F_8\}$, 14341 for $V = \{F_{11}\}$, and 13169 for $V = \{F_{14}\}$. 
This suggests to start the search by contacting a finger with a high index (e.g., $F_{14}$), when it is known that the resource is “rare.” When there is no information about the popularity of the resource to be found, an intermediate finger (e.g., $F_{11}$) should be used. As shown in Figure 7b, also the search time decreases as the replication rate increases, for any value of $V$. When $V = \{F_8\}$, the average search time passes from 22.3 for $r = 0.25\%$, to 16.1 for $r = 32\%$. When $V = \{F_{14}\}$, the search time ranges from 24.4 for $r = 0.25\%$, to 5.2 for $r = 32\%$.

The graph shows that with low values of $r$ it is convenient to contact a finger with a high index, which leads to a lower search time with respect to fingers with a lower index. However, since the main objective of DQ-DHT is reducing the number of messages, an intermediate finger (e.g., $F_{11}$) should be preferred in most cases, even if this may result to an increased search time.

We ran a second set of simulations to evaluate the effect of varying the initial value of $L$. According to the results discussed above, we chose an intermediate finger for the probe query ($V = \{F_{11}\}$), and varied $L$ from 2 to 8, with $R_d$ fixed to 100. The results are presented by the graphs in Figure 8.

Figure 8b shows that lower values of $L$ generate lower search times. For instance, when $r = 1\%$ the average search time passes from 17.1 with $L = 2$, to 25.2 with $L = 8$. This is mainly due to the fact that the wait after the probe phase is proportional to $L$, as described in Section 2.2.

On the other hand, Figure 8a shows that very low values of $L$ produce a significant increase in the number of messages. For example, when $r = 1\%$ the average number of messages passes from 14259 with $L = 8$, to 34654 with $L = 2$. The excess of messages in the second case is due to the reduced accuracy in the estimation of the resource popularity that is obtained considering only a few levels of the subtrees associated to $V$.

In general, intermediate values of $L$ produce the best compromise between number of messages and search time. For the scenario analyzed here ($V = \{F_{11}\}$), the best result is obtained with $L = 4$, which generates a number of messages similar to that produced by higher values of $L$, but with a quite lower search time, as shown by the graphs in Figure 8.
4.2. **DQ-DHT over a k-ary DHT**

For the $k$-ary version of DQ-DHT different combinations of the algorithm parameters $H_P$ and $H_E$ have been experimented, with $R_d$ fixed to 100.

We ran a first set of simulations in a $k$-ary DHT with $k = 2$ (i.e., a Chord network), with $H_P$ fixed to 2000, and $H_E$ ranging from 250 to 2000. The goal of this first set of experiments was evaluating the behavior of the algorithm (i.e., number of messages and search time) while varying the number $H_E$ of nodes that have received the query before to estimate the resource popularity.

The graphs in Figure 9 show number of messages and search time (expressed in time units) as a function of the replication rate. Figure 9a shows that the number of messages decreases when the replication rate increases, for any value of $H_E$, as in the Chord-based version. In general, the number of messages is lower for higher values of $H_E$. In fact, the generated number of messages depends on the accuracy of the popularity estimation, which is better when $H_E$ is higher. This is particularly true in presence of low replication rates. For example, the number of messages for $r = 0.5\%$ passes from 25889 with $H_E = 2000$, to 31209 with $H_E = 250$.

![](image1.png)  
**Figure 9.** Effect of varying the value of $H_E$, with $H_P = 2000$ and $k = 2$: (a) number of messages; (b) search time.

As shown in Figure 9b, also the search time decreases as the replication rate increases. Moreover, the search time decreases as the value of $H_E$ decreases, since lower values of $H_E$ correspond to a lower duration of the probe query. For instance, the search time for $r = 0.5\%$ passes from 29.58 with $H_E = 2000$, to 22.53 with $H_E = 250$. However, since lower values of $H_E$ generate more messages, an intermediate value of $H_E$ should be preferred. For example, $H_E = 1000$ represents a good compromise since it generates the same number of messages of $H_E = 2000$, but with a search time close to that of $H_E = 250$.

Then, we compared the performance of the algorithm with different values of $k$. Based on the first set of simulations, we chose the following algorithm parameters: $H_P = 2000$ and $H_E = 1000$. Figure 10 shows how number of messages and response time vary in this configuration with $k$ ranging from 2 to 8.

As shown in Figure 10b, the search time strongly depends on the arity of the DHT. The maximum gain (nearly 48\%) is obtained for $r = 0.5\%$, with the search time passing
from 24.46 with $k=2$, to 12.74 with $k=8$. The minimum gain (20 %) is obtained for the highest replication rate ($r = 32 \%$), when the search time passes from 5.02 with $k=2$, to 4.0 with $k=8$. The number of messages is less related to the value of $k$ than the search time (see Figure 10a), but - in general - lower values of $k$ generate lower number of messages. The maximum difference between $k=2$ and $k=8$ is reached with $r = 0.5 \%$ (about 14 %), but it is counterbalanced by a search time gain of 48 %, as shown in Figure 10b.

We repeated the comparison above using the following configuration: $H_P = 4000$ and $H_E = 2000$. Since $H_P$ is the minimum number of messages that will be generated during the search process, a so high value should be used when it is fundamental to minimize the search time. The simulation results are reported in Figure 11.

The trends are similar to those shown in Figure 11. In general, the search time is lower by 1-2 units w.r.t. that measured for $H_P = 2000$ and $H_E = 1000$. For $r = 4 \%$, the search time is significantly improved because the probe query, with $H_P = 4000$, resulted in most cases sufficient to obtain the desired number of results.
In summary, the simulation results presented above demonstrate that implementing dynamic querying over a $k$-ary DHT allows to achieve a significant improvement of the search time with respect to the Chord-based implementation.

5. Related Work

The work most related to DQ-DHT is the Structella system designed by Castro et al. [19]. Structella replaces the random graph of Gnutella with the structured overlay of Pastry [20], while retaining the content placement and discovery mechanisms of unstructured P2P systems to support complex queries. Two discovery mechanisms are implemented in Structella: constrained flooding and random walks.

Constrained flooding is based on the algorithm of broadcast over Pastry presented in [21]. A node $x$ broadcasts a message by sending the message to all the nodes $y$ in the Pastry’s routing table. Each message is tagged with the routing table row $r$ of node $y$. When a node receives a message tagged with $r$, it forwards the message to all nodes in its routing table in rows greater than $r$. To constrain the flood, an upper bound is placed on the row number of entries to which the query is forwarded.

Random walks in Structella are implemented by walking along the ring formed by neighboring nodes in the identifier space. When a node receives a query in a random walker, it uses the Pastry’s leaf set to forward the query to its left neighbor in the identifier space. It also evaluates the query against the local content and sends matching content back to the query originator. A random walker is terminated when it finds matching content. Multiple concurrent random walkers can be used to improve search time.

DQ-DHT and Structella share the same goal of supporting complex queries in structured network. However, DQ-DHT has been designed to find an arbitrary number of resources matching the query criteria, while Structella is designed to discover just one of such resources. In Structella in fact, with both constrained flooding and random walks, a node stops forwarding a query if it has matching content. This functional difference makes DQ-DHT and Structella not comparable, so we cannot provide a comparison of their performance.

A way to let Structella return an arbitrary number of results instead of just one could be modifying its random walks algorithm, using the same termination mechanisms proposed for random walks in unstructured networks [22]. Unfortunately, a direct interaction between querying node and walkers may be infeasible in some networks (e.g., due to firewalls), and generates overload of the querying node if too many walkers are used or the communication with them is too frequent [22]. It is worth noticing that DQ-DHT, on the contrary, does not require any remote interaction to terminate the search.

A few other research works broadly relate to our system for their combined use of structured and unstructured P2P techniques.

Loo et al. [23] propose a hybrid system in which DHT-based techniques are used to index and search rare items, while flooding techniques are used for locating highly-replicated content. Search is first performed via conventional flooding techniques of the overlay neighbors. If not enough results are returned within a predefined time, the query is reissued as a DHT query. This allows fast searches for popular items and at the same time reduces the flooding cost for rare items.

A critical point in such system is identifying which items are rare and must be published using the DHT. Two techniques are proposed. A first heuristic classifies as rare the
items that are seen in small result sets. However, this method fails to classify those items that have not have been previously queried and found. Another proposal is to base the publishing on well-known term frequencies, and/or by maintaining and possibly gossiping historical summary statistics on item replicas.

Another example is the work by Zaharia and Keshav [24], who focus on the problem of selecting the best algorithm to be used for a given query in a hybrid network allowing both unstructured search and DHT-based lookups. A gossip-based algorithm is used to collect global statistics about document availability and keyword popularity that allow peers to predict the best search technique for a given query.

Each peer starts by generating a synopsis of its own document titles and keywords and labels it as its “best” synopsis. In each round of gossip, it chooses a random neighbor and sends the neighbor its best synopsis. When a node receives a synopsis, it fuses this synopsis with its best synopsis and labels the merged synopsis as its best synopsis. This results in every peer getting the global statistics after \( O(\log N) \) rounds of gossip.

Given a query composed by a set of keywords, a peer estimates the expected number of documents matching that set of keywords using the information in its best synopsis. If this number is over a given threshold, many matches are expected, so the peer floods the query. Otherwise, it uses the DHT to search for each keyword, requesting an in-network join, if that is possible. The flooding threshold is dynamically adapted by computing the utility of both flooding and DHT search for a randomly chosen set of queries.

It is worth noticing that the last two systems do not support arbitrary queries, since information about resources is published and searched using DHT-based mechanisms. DQ-DHT, on the contrary, supports arbitrary queries in an easy way since content placement is unrelated to the DHT overlay and query processing is performed on a node-by-node basis.

6. Conclusions

Information services are key components of Grid systems as they provide the basic mechanisms to index and search the resources needed to run distributed applications. To implement scalable information services in large-scale Grids, several DHT-based P2P systems have been proposed. Those systems support efficient search of resources based on some predefined attributes, but do not support arbitrary queries, like regular expression, which can be necessary to select resources based on complex criteria or semantic features.

We focused on designing a P2P search algorithm, named DQ-DHT, to support arbitrary queries over DHT-based overlays. This algorithm has been particularly designed with the aim of extending existing DHT-based Grid information services with the capability of performing arbitrary queries. In this chapter we described the DQ-DHT algorithm using Chord as basic overlay. We also described an extension of DQ-DHT allowing to perform dynamic querying search in a \( k \)-ary DHT-based overlay.

The simulation results demonstrated that DQ-DHT dynamically adapts the search extent based on the popularity of the resource to be located and the desired number of results, also allowing to control its performance (i.e., search time versus number of messages) based on application requirements and system objectives.
References


Emulation platform for high accuracy failure injection in grids

Thomas HERAULT\textsuperscript{a} Mathieu JAN\textsuperscript{b} Thomas LARGILLIER \textsuperscript{a,1} Sylvain PEYRONNET\textsuperscript{a} Benjamin QUETIER \textsuperscript{a} Franck CAPPELLO \textsuperscript{c}

\textsuperscript{a}Univ Paris Sud-XI FR-91405; LRI; INRIA Saclay; CNRS
\textsuperscript{b}CEA, LIST, FR-91191
\textsuperscript{c}INRIA Saclay FR-91893; Univ Paris Sud-XI; LRI; CNRS

Abstract. In the process of developing grid applications, people need to often evaluate the robustness of their work. Two common approaches are, simulation where one can evaluate his software and predict behaviors under conditions usually unachievable in a laboratory experiment and experimentation where the actual application is launched on an actual grid. However, simulation could ignore unpredictable behaviors due to the abstraction done and experimentation does not guarantee a controlled and reproducible environment.

In this chapter, we propose an emulation platform for parallel and distributed systems including grids where both the machines and the network are virtualized at a low level. The use of virtual machines allows us to test highly accurate failure injection since we can “destroy” virtual machines and, network virtualization provides low-level network emulation. Failure accuracy is a criteria that notes how realistic a fault is. The accuracy of our framework is evaluated through a set of micro benchmarks and a very stable P2P system call Pastry since we are very interested in the publication system and resources finding of grid systems.

Introduction

One of the most important issues for the evaluation of a grid application is to monitor and control the experimental conditions under which this evaluation is done. This is particularly important when it comes to reproducibility and analysis of observed behavior. In grid software systems, the experimental conditions are diverse and numerous, and can have a significant impact on the performance and behavior of these systems. As a consequence, it is often very difficult to predict from theoretical models what performance will be observed for an application running on a large, heterogeneous and distributed system. It is thus often necessary and insightful to complement the theoretical evaluation of parallel algorithms with simulations and experiments in the “real world”.

However, even with detailed monitoring procedures, experiments in the real world are often subject to the influence of external events, which can prevent more detailed analysis. More importantly, the experimenters usually have access to only a small variety of distributed systems. In general, experimental conditions are not strictly reproducible in

\textsuperscript{1}Corresponding author: Université Paris Sud XI, Bâtiment 490 91405 Orsay Cedex France; Email: thomas.largillier@lri.fr
the real world. The approach usually taken to broaden the scope of the evaluation consists in designing simulators, under which the experimental conditions can be as diverse as necessary. The “real world” experiments can then help to validate the results given by simulators under the reproduced similar conditions.

Still, simulators can only handle a model of the application, and it is hard to validate an implementation, or guarantee that the end user application will meet the predicted performance and behavior. Here, we study another tool for experimentations: emulators. Emulators are a special kind of simulator, which are able to run the final application, under emulated conditions. They do not make the same kind of abstraction as normal simulators, since they emulate the hardware parts of all the components of the real world infrastructure, and thus capture the complex interactions of software and hardware. Yet, since the hardware is emulated in software, the experimenter has some control on the characteristics of the hardware used to run the application.

Through this control, the experimenter can design an ad-hoc system, suitable for his experiments. Of course, the predicted performances must still be validated by comparison with experiments on real world systems, when such systems exist. But within an emulated environment, the experimenter can inject experimental conditions that are not accessible in a real environment, or not controlled. A typical example of such condition is the apparition of hardware failures during the experiment. With a real system, hardware failures are hard to inject, and hard to reproduce. In an emulated environment, hardware is software-controlled and the experimenter can design a reproducible scenario of fault injection to stress fault tolerant applications. This is crucial in fault-tolerant systems, since the impact of the timing and target of a failure can impact tremendously on the liveness and performance of the application.

Emulators can be designed at different levels of the software stack. A promising approach for emulators is the use of virtual machines (VM). A VM by itself fits partially the goals of parallel application emulators, since it emulates (potentially multiple) instances of a virtual hardware on a single machine. In addition to these virtual machines, we need to link them through a controllable network. In this chapter, we present V-DS, a platform for the emulation of parallel and distributed systems (V-DS stands for Virtual Distributed System) through virtualization of the machines and the network.

V-DS introduces virtualization of all the hardware of the parallel machine, and of the network conditions. It provides to the experimenter a tool to design a complex and realistic failure scenario, over arbitrary network topologies. To the best of our knowledge, this is one of the first systems to virtualize all the components of a parallel machine, and provide a network emulation that enables experimenters to study low-level network protocols and their interactions with failures.

This work is an extended version of the short paper presented at the ACM International Conference on Computing Frontiers 2009 in the paper entitled “High accuracy failure injection in parallel and distributed systems using virtualization”[14]. This chapter presents with more details the performance analysis that let us conclude to a better accuracy of failure injection using our tool, when compared with other tools, with new micro-benchmarks and experiments completing the previous results.

This chapter is organized as follows. Section 1 presents related work. Then, we give the design of our platform in section 2. Section 3 presents the experiments we made for testing the platform. Finally we conclude and introduce future works in the last section.
1. Related Work

Recently, the number of large-scale distributed infrastructures has grown. However, these infrastructures usually fall either into the category of production infrastructures, such as EGEE\(^2\) or DEISA [17], or in the category of research infrastructures, such as PlanetLab [10]. To our knowledge, only one of the testbeds in the latter category, namely Grid’5000 [6], meets the mandatory requirement for performing reproducible and accurate experiments: full control of the environment.

For instance, PlanetLab [10] is a good example of testbed lacking the means to control experimental conditions. Nodes are connected over the Internet, and a low software reconfiguration is possible. Therefore, PlanetLab depends on a specific set of real-life conditions, and it is difficult to mimic different hardware infrastructure components and topologies. Consequently, it may be difficult to apply results obtained on PlanetLab to other environments, as pointed out by [13]. Grid’5000 [6] consists of 9 sites geographically distributed through France. It is an example of a testbed which allows experiments in a controlled and precisely-set environment. It provides tools to reconfigure the full software stack between the hardware and the user on all processors, and reservation capabilities to ensure controllable network conditions during the experiments. However, much work remains to be carried out for injecting or saving, in an accurate and automatic manner, experimental conditions in order to reproduce experiments. Finally, Emulab [24] is an emulation platform that offers large-scale virtualization and low-level network emulation. It integrates simulated, emulated and live networks into a common framework, configured and controlled in a consistent manner for repeated research. However, this project focuses only on the full reconfiguration of the network stack. Moreover, Emulab uses extended FreeBSD jails as virtual machines. Inside jails, the operating system is shared between the real machine and the virtual machine, thus killing a virtual machine is the same as killing a process. It means that this framework may not simulate real (physical) machine crashes. To the contrary, our work uses Xen virtual machines, allowing to either shutdown the machine or crash it, which will leave the connections open. As will be demonstrated in the experiments section, this is a much more realistic crash simulation since a crashed machine never closes its connections before disappearing from the network.

Software environments for enabling large-scale evaluations most closely related to ours are [4] and [18]. [4] is an example of integrated environment for performing large-scale experiments, via emulation, of P2P protocols inside a cluster. The proposed environment allows the experimenter to deploy and run 1000000 peers, but at the price of changes in the source codes of tested prototypes and supporting only Java-based applications. Besides, this work concentrates on evaluating the overhead of the framework itself and not on demonstrating the strength of it by, for instance, evaluating P2P protocols at large scales. In addition, the project provides a basic and specific API suited for P2P systems only. P2PLab [18] is another environment for performing P2P experiments at large-scale in a (network) controlled environment, through the use of Dummynet [20]. However, as for the previously mentioned project [4], it relies on the operating system scheduler to run several peers per physical node, leading to CPU time unfairness. Modelnet [23] is also based on Dummynet. It uses the same scheme except that the network

control nodes do not need to co-scheduled on the compute nodes. In Modelnet, network nodes are called *core nodes* and compute nodes *edge nodes*. But, as in P2PLab, multiple instances of applications are launched simultaneously inside *edge nodes*, consequently it relies again on the operating system to manage several peers.

Virtualization in Clouds and emulators serve different objectives: In Clouds, virtualization allows many users sharing the same hardware. In our emulation engines, virtualization is used to run, for a single user, many instances of virtual nodes on the same hardware. One of the main goal of the virtualization systems used in Clouds is to ensure that no data or program in one virtual machine can be accessed and corrupted from another virtual machine running on the same hardware. Thus virtual machine security (isolation) is a major concern. In contrary a virtualization environment for emulation should make easy the communication between virtual machines and since the same user is using all virtual machines, there is no need for security. In Emulation, one goal is to run the maximum number of virtual machines on the same hardware. This goal is not considered as essential for Clouds. These differences in goals are fundamental and as a consequence, the virtualization technologies developed for Clouds are not corresponding to the need of Emulators.

Finally, simulators, like Simgrid [7], GridSim [5], GangSim [11], OptorSim, [2], etc. are often used to study distributed systems. The main problem with simulation is that it successfully isolates protocols but does this at the expense of accuracy. Some problems that have been overlooked by the abstractions done in the simulation will not be exhibited by simulations but will be observed when the real application is launched, so conclusions from simulation may not be valid, like in [12].

### 2. V-DS Platform Description

The V-DS virtualization environment is composed of two distinct components: the virtualization environment for large-scale distributed systems and a BSD-module for the low-level network virtualization. Each component will be described in the following subsections.

#### 2.1. Virtualization Environment for Large-scale Distributed Systems

V-DS virtualizes distributed systems entities, at both operating and network level. This is done by providing each virtual node its proper and confined operating system and execution environment.

V-DS virtualizes a full software environment for every distributed system node. It allows the accurate folding of a distributed system up to 100 times larger than the experimental infrastructure [19], typically a cluster.

V-DS supports three key requirements:

- Scalability: In order to provide insights on large-scale distributed systems, V-DS supports the folding of distributed systems. Indeed, thanks to Xen characteristics, it is possible to incorporate a large number of virtual machines on a single physical machine with a negligible overhead [1].
• Accuracy: In order to obtain accurate behavior of a large-scale distributed system several constraints on the virtual machines (VMs) are needed. First the CPU must be fairly shared between VMs, then each VM must be isolated from the others, lastly the performance degradation of a VM must evolve linearly with the growth of the number of VMs. Using Xen allows V-DS to ensure all these requirements (see for example [19]).

• Adaptivity: the platform provides a custom and optimized view of the physical infrastructure used. For instance, it is possible to support different operating systems, and even different versions of the same operating system.

V-DS is based on the Xen [1] virtualization tool in version 3.2. Xen gets interesting configuration capabilities, particularly at the network level which is fundamental in the injection of network topologies. Compared to other virtualization technologies it has been demonstrated [19] that Xen offers better results.

Figure 1 shows the general architecture of V-DS. Here, $m$ physical machines called $PM-i$ running a Xen system are hosting $n$ virtual machines named $VM-i-j$ with $i$ the index of the physical machine hosting that virtual machine and $j$ the index of the virtual machine. Thus, there are $n \times m$ virtual machines (VM). All communications between these VM are routed to FreeBSD machines to 1) prevent them from communicating directly through the internal network if they are on the same physical machine, 2) add network topologies between VM.

2.2. Low-level Network Virtualization

One of the main advantages of the V-DS platform is that it also uses virtualization techniques for emulating the network. This allows the experimenter to emulate any kind of
topology with various values for latency and bandwidth on a cluster. For instance, we can run, in this framework, grid applications on clusters.

For the purpose of emulating the network, we use FreeBSD machines. Using BSD machines to virtualize the network is crucial to a reliably accurate network simulation, since BSD contains several very efficient tools to manipulate packages like ipfw\(^3\) or Dummynet\(^{[20]}\). With these packages it is possible to insert failure (like dropping packets) in the network very easily. The platform is then capable of injecting realistic failures at the machine and network level.

There are three networks joining the virtual machines. The first is a classic ethernet network. Each virtual machine has its own ethernet card. The second one uses Myrinet cards and provides very fast links between the nodes. The last one offers layer 2 virtualization using the EtherIP protocol \(^{[16]}\). EtherIP bridges are set between any virtual machine and the corresponding BSD machine.

To set up the bridges, we use a topology file given by the user. The format we use for the topology is the dot format \(^4\) which is easy to manipulate and to write. There are two different types of nodes for the topology. The Xen nodes representing the virtual machines and the BSD nodes representing routers. The dot language being very simple it is easy to generate well-known topologies such as rings, cliques, etc. As the language is well-spread, there also exist graphical tools to design specific topologies.

Using the topology file, we generate the routing table of each BSD machine. The routing is made through a kernel module. More precisely it is a netgraph\(^5\) node called “ip\_switch”. This node works with ipfw, allowing the user to filter packets and to redirect them into a netgraph node. Here we filter all EtherIP packets.

These packets are examined by the module who stores its routing table in a kernel hash table. After modifying the IP header of the packet to correctly route it to the next hop, the packet is put back on the network. The packet may also enter a Dummynet rule before or after being rerouted. The module can also deal with ARP (Address Resolution Protocol) requests in which case it will forward the request to all his neighbors.

3. Experiments

In this section we present the experiments we perform in order to assess the performances and functionalities of our virtualization framework. All these experiments were done on Grid\textsuperscript{5000} \(^{[6]}\). Grid\textsuperscript{5000} is a computer science project dedicated to the study of grids, featuring 13 clusters, each with 58 to 342 PCs, connected by the Renater French Education and Research Network. For our experiments we used a 312-node homogeneous cluster composed of AMD Opteron 248 (2.2 GHz/1MB L2 cache) bi-processors running at 2GHz. Each node feature 20GB of swap and SATA hard drive. Nodes were interconnected by a Gigabit Ethernet switch. All our experiments were performed using a folding ratio of 10 (e.g. each physical node runs 10 virtual machines).

All the following experiments ran under Xen version 3-2, with Linux-2.6.18.8 for the Physical and Virtual Computing nodes, and BSD version 7-OPRERELASE for the network emulation. Since we were embedding a Java virtual machine on the Xen virtual

\(^{3}\text{http://www.freebsd.org/doc/en/books/handbook/firewalls-ipfw.html}\)

\(^{4}\text{http://www.graphviz.org/doc/info/lang.html}\)

\(^{5}\text{http://people.freebsd.org/~julian/netgraph.html}\)
Table 1. Respect of network restraint conforming measures

<table>
<thead>
<tr>
<th>Requested value</th>
<th>Measured value - w/o NE</th>
<th>Measured value - with NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 Mbps</td>
<td>241.5 Mbps</td>
<td>235 Mbps</td>
</tr>
<tr>
<td>25 Mbps</td>
<td>24.54 Mbps</td>
<td>23.30 Mbps</td>
</tr>
<tr>
<td>2.5 Mbps</td>
<td>2.44 Mbps</td>
<td>2.34 Mbps</td>
</tr>
<tr>
<td>256 Kbps</td>
<td>235.5 Kbps</td>
<td>235.5 Kbps</td>
</tr>
</tbody>
</table>

Table 1. Respect of network restraint conforming measures

(a) Bandwidth restraint

<table>
<thead>
<tr>
<th>Requested value</th>
<th>Measured value - w/o NE</th>
<th>Measured value - with NE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 ms</td>
<td>10.1 ms</td>
<td>10.2 ms</td>
</tr>
<tr>
<td>50 ms</td>
<td>52.2 ms</td>
<td>52.1 ms</td>
</tr>
<tr>
<td>100 ms</td>
<td>100.2 ms</td>
<td>100.4 ms</td>
</tr>
<tr>
<td>500 ms</td>
<td>500.4 ms</td>
<td>500.8 ms</td>
</tr>
</tbody>
</table>

(b) Latency restraint

machines we needed it to be light. We chose the 1.5.0,10-eval version that fulfilled our needs and our space requirements.

3.1. Impact of the Low-Level Network Emulation

We first measured the impact of the network emulation of V-DS on the network bandwidth and latency, using the netperf tool [22]. To do this, we used two version of V-DS: with network emulation at the high level only (when packets are slowed down by the router, but not encapsulated in an IP over ethernet frame), and with low-level network emulation, as described in section 2.

The experimental setup consisted in three physical machines: one running the BSD router, the other two running one virtual machine each. We configured the BSD router to introduce restraints on the network, either using low-level emulation with ethernet over IP, or without low-level emulation.

The results are summed up in table 1. The requested value represents the restraint imposed by the virtualization. In this table low-level network emulation is denoted as NE.

Regarding the bandwidth, the obtained values are very close to the requested ones, the difference being around 3%. This corresponds to the time spent in the traversing of the virtual layer. Netperf tests are realised at the TCP level, implying that part of the bandwidth is used for the TCP protocol.

When adding the low-level network emulation the bandwidth drops again for another 3%. This could be explained by the encapsulation needed by the etherip protocol for a full network emulation. There is no significant difference regarding the latency measures with the low-level emulation.

3.2. TCP Broken Connection Detection Mechanism

In this set of experiments, we stress the broken connection detection mechanism implemented in the TCP stack. Many applications rely on TCP detection mechanism to detect
failures and implement their own fault-tolerance strategy, thus the efficiency of TCP failure detection has a significant impact on the efficiency of these applications. The failure detection mechanism of TCP relies on heartbeats, under the so-called pull model [3]: one peer sends a heartbeat to the other peer, and starts a timer; when a peer receives a heartbeat, it will send an acknowledgement back; if the acknowledgement returns before the expiration of the timer, the sending peer assumes that the receiving peer is alive; if the timer expires before the reception of the acknowledgement, the connection is broken.

This mechanism is controlled at the user level through BSD socket parameters: $SO_{\text{KEEPALIVE}}$ enables the failure detection mechanism, which is tuned through $\text{tcp}._{\text{keepalive}}._{\text{time}}, \text{tcp}._{\text{keepalive}}._{\text{probes}},$ and $\text{tcp}._{\text{keepalive}}._{\text{intvl}}$. $\text{tcp}._{\text{keepalive}}._{\text{time}}$ defines how long a socket can be without traffic before beginning the heartbeat protocol; $\text{tcp}._{\text{keepalive}}._{\text{probes}}$ defines the number of heartbeats that can be lost on a socket before the connection is considered to be broken; $\text{tcp}._{\text{keepalive}}._{\text{intvl}}$ defines the maximum time to wait before considering that a heartbeat has been lost.

To stress the failure detection mechanism of TCP, we designed three simple synthetic benchmarks. They all assume a single pair of client/server processes connected through TCP BSD sockets. In the first benchmark ($\text{Send}$), the client sends messages continuously to the server without waiting for any answer. In the second benchmark ($\text{Recv}$), the client waits for a message from the server. In the third ($\text{Dialog}$), the client and server are alternatively sending and receiving messages to/from each other.

In all those experiments the server is killed or destroyed right after the connection is established and we measure the elapsed time before the client realizes the connection has been broken. We set the $\text{tcp}._{\text{keepalive}}._{\text{time}}$ to 30 minutes, the $\text{tcp}._{\text{keepalive}}._{\text{probes}}$ to 9 and the $\text{tcp}._{\text{keepalive}}._{\text{intvl}}$ to 75 seconds, which are the default on Linux machines (except for the keepalive time, which was reduced to lower the duration of the experiments). As a consequence, when a machine crashes we expect the other side to notice the event within a period of approximately 41 minutes.

We then have two sets of experiments, one where the server process is killed and one where the machine hosting the process is destroyed. Each set of experiments includes the three benchmarks in both Java and ANSI C. Every benchmark is run twice, once where the $SO_{\text{KEEPALIVE}}$ variable is on and once where it’s off. The results are summed up in table 2.

<table>
<thead>
<tr>
<th>Failure Injection Method</th>
<th>Language</th>
<th>Socket Option</th>
<th>Send</th>
<th>Receive</th>
<th>Dialog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kill</td>
<td>C</td>
<td>-</td>
<td>0.2s</td>
<td>0.3s</td>
<td>0.2s</td>
</tr>
<tr>
<td>Kill</td>
<td>Java</td>
<td>-</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Destroy</td>
<td>C</td>
<td>$SO_{\text{KEEPALIVE}}$</td>
<td>17min</td>
<td>41min</td>
<td>15min30s</td>
</tr>
<tr>
<td>Destroy</td>
<td>Java</td>
<td>$SO_{\text{KEEPALIVE}}$</td>
<td>$\infty$</td>
<td>41min</td>
<td>15min30s</td>
</tr>
<tr>
<td>Destroy</td>
<td>C</td>
<td></td>
<td>17min</td>
<td>$\infty$</td>
<td>15min30s</td>
</tr>
<tr>
<td>Destroy</td>
<td>Java</td>
<td></td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>15min30s</td>
</tr>
</tbody>
</table>

Table 2. TCP failure detection times

In this table, the value $\infty$ means that after a long enough amount of time (several hours) exceeding significantly the expected time of the failure detection (41 minutes) the active computer has still not noticed that the connection has been broken. The N/A value means that the language or the system does not notify errors even if it detects
a broken link. In the case of Failure Injection with the Kill method, the socket option SO_KEEPALIVE has no effect on the results.

When using the Kill failure injection method, one can see that the Linux operating system detects the failure at the other end almost instantaneously. This is due to the underlying TCP/IP protocol: the process is killed, but the operating system continues to work, so it can send the RST packet to the living peer, which will catch it and notify the process of a “failure”. For the Java virtual machine, the socket is also notified as closed, but the language does not notify this closure as a failure: the code also has to check continuously for the status of the Input and Output streams, in order to detect that a stream was unexpectedly closed. In our JVM implementation, no exceptions were raised when sending on such a stream, and receptions gave null messages.

On the contrary, when using a more realistic destroy mechanism, the operating system of the “dead” peer is also destroyed. So, no mechanism sends a message to the living peer to notify of this crash. The living peer must rely on its own actions to detect failures, which is a more realistic behavior. We distinguish between the two cases when the SO_KEEPALIVE option is either on or not on the socket. In native Linux applications (ANSI C programs), the failure is always detected when the SO_KEEPALIVE option is on. TCP also uses the communications induced by normal traffic to detect a potential failure, that is why the detection time is lower for the Dialog and Send benchmarks.

In the case of the Recv benchmark, the living peer does not introduce communication in the network, so the system has to rely on the heartbeat procedure, which uses conservative values to detect the failure with a low chance of false positives, and a small perturbation of the network.

It is clear from these experiments that crash injection through complete destruction of the virtual machine, including the operating system, exhibit more accurate behavior than the simple destruction of a process, even using a forced kill method, because the underlying operating system will clean up the allocated resources, including the network resources.

3.3. Stress of Fault-Tolerant Applications

In order to evaluate the platform capabilities to inject failures, we stressed FreePastry which is an open-source implementation in Java of Pastry [21,9] intended for deployment on the Internet. Pastry is a fault-tolerant peer-to-peer protocol implementing distributed hash-tables. In Pastry every node has a unique identifier which is 128 bits long. This identifier is used to position the node on a $2^{128}$-place oriented ring. A key is associated to any data, using a hash function, and each process of identifier $id < id'$ (where $id'$ is the identifier of the next process on the ring) holds all data with key $k$ such that $id \leq k < id'$. Then, by comparing the process identifiers and data keys, any process can route any message to a specific data. Shortcuts between nodes (called fingers in Pastry) are established to ensure logarithmic time to locate a node holding any data from any other node.

When a node is joining an existing ring, it gets a node id and initializes its leaf set and routing table by finding a “close” node according to a proximity metric. Then it asks this node to route a special message with its identifier as a key. The node at the end of the road is the one with the closest identifier and then the new node takes its leaf set and its routing table is updated with information gathered along the road. The new node will
then send messages in the pastry network to update the routing table of all processes it should be connected to.

Pastry manages nodes failures as nodes departures without notification. In order to handle this kind of situation “neighbors” (nodes which are in each others leaf set) exchange keepalive messages. If a node is still not responding after a period T, it is declared failed and everyone in its leaf set is informed. The routing table of all processes that the departing process was connected to are then updated. This update procedure can take some time and is run during the whole life of the distributed hash table. At some point in time, the routes stop changing (they are stabilized), but the maintaining procedures for these routes continue to execute.

In order to validate the platform we looked at three things. First we evaluated the average time for the system to stabilize itself after all the peers had joined the network. Then we evaluated the average time needed for every node to know that a node was shut down or killed. In the first case we only kill a java process and in the second we “destroy” the virtual machine which is hosting the process.

The experiments go as follows. The first virtual machine (called the bootstrap node) creates a new ring and then every other virtual machine connects to it. We ask every node for its routing table every 200ms and log it whenever it changes together with a time stamp.

In order not to overwhelm the bootstrap node, we launch machines by groups of tens separated by a 1 second interval. The results for the first experiment are presented in figure 2 below.

![Figure 2. Average number of changes left by machines](image-url)
It can be seen that for even small rings, composed of as few as 50 machines out of a possible $2^{128}$, the time for the system to stabilize is huge (over 5 hours). This time increases with the numbers of machines and can still be over 18h for a ring as small as 400 machines.

To reduce the duration of the experiments, we made use of the fact that a majority of changes in the routing tables are made in the first few seconds of initialization. It appears that after only 100s more than 50% of the changes have been made. Thus we do not wait for the whole system to be stabilized before injecting the first failure, but we wait for the whole system to have made enough changes in the routing tables and for it to be in a relatively steady state. The first failure is injected 45min after the beginning of the experiment.

We call D-node the node we suppress from the ring, either by killing the process or destroying the machine. After suppressing the D-node we wait for 20 min for the nodes to update their routing tables. After this period we collect the routing tables and look for those which include the D-node. In those particular tables we search for the update that will make the D-node disappear from the routing table.

![D-node deletion time](image)

**Figure 3.** D-node deletion time

Figure 3 presents the cases when we “destroy” the virtual host of the process, and when we kill the process. Each dot in this figure represents the update of the routing table of process $y$, at a time $x$, concerning the D-node. The circles represent the modifications before the failure is injected, thus modifications due to the normal stabilization of FreePastry. The squares represent the modifications after the injection of the failure for the D-node in the case of process kill, and the triangles in the case of virtual host
destruction. The vertical line represents the date of the failure injection at the D-node (45 minutes after the beginning).

The set of routing tables that include the D-node consists of 578 nodes over several experiments. In this set many nodes delete the D-node of their routing table before it is suppressed. As it can be seen on the figure, all these nodes do it very early in the stabilization and therefore we can consider that every node that deletes the D-node from its table after the suppression time does it thanks to the failure detection component of Pastry.

Since the routing table maintenance is done lazily in Pastry [8], it is natural that not every node updates its routing table, since in the experiments no messages are exchanged. When we only kill the pastry process to suppress the D-node after 45 minutes we can see on figure 3 that a lot of nodes react in a very short period of time to the suppression of the D-node. Comparing the points distributions for Kill and Destruction, we can see that nodes detect the failure in a shorter period of time in the case of kill than in the case of destruction. Since behaviors in the two cases is different we can consider that “destroying” a machine is more accurate since the stressed application must rely on its own failure detection mechanism, and the behavior of this application may be influenced by the asynchronism and the timings of the failure detection mechanism used. The figure also demonstrates that the active failure detection mechanism of FreePastry is effective and the distributed hash table is able to stabilize even with accurate failure injection.

Conclusion and Future Work

In this chapter, we presented an emulation platform for grids where both the machines and the network are virtualized at a low level. This allows an experimenter to test realistic failure injection into applications running on distributed architectures, such as grids. We evaluated the interest of our approach by running a classical fault-tolerant distributed application: Pastry.

We are in the process of developing a fault injection tool to work with the platform. It will be an extension of the work started in the tool Fail [15]. The interest of this work is that using Xen virtual machines will allow to model strong adversaries since it is possible to have virtual machines with shared memory. These adversaries will be stronger since they will be able to use global fault injection strategies.

Part of this work is already available on the web\(^6\) and a tutorial is also available online\(^7\). This version does not include the layer 2 network virtualization because it is not packaged yet and will be available as soon as possible.

Acknowledgements. Experiments presented in this chapter were carried out using the Grid’5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners.

\(^6\)http://www.lri.fr/~quetier/v-ds/v-ds-1.4.tgz
\(^7\)http://www.lri.fr/~quetier/vgrid/tutorial
References


Abstract. This paper presents an overview of the DEISA2 project, vision, mission, objectives, and the DEISA infrastructure and services offered to the e-science community. The different types of applications are discussed which specifically benefit from this infrastructure and services, and the DEISA Extreme Computing Initiative for supercomputing applications is highlighted. Finally, we analyse the DEISA sustainability strategy and present lessons learned.

Keywords. DEISA, High Performance Computing, Grid Computing, Applications

Introduction

The DEISA Consortium has deployed and operated the Distributed European Infrastructure for Supercomputing Application (DEISA, [4]), co-funded through the EU FP6 DEISA project from 2004 to 2008. Since May 2008, the consortium continues to support and further develop the distributed high performance computing infrastructure and its services through the EU FP7 DEISA2 project with funds for another three years until 2011. Activities and services relevant for applications enabling, operation, and technologies are continued and further enhanced, as these are indispensable for the effective support of computational sciences in the HPC area. The resulting infrastructure is unmatched world-wide in its heterogeneity and complexity, enabling the operation of a powerful Supercomputing Grid built on top of national services, facilitating Europe’s ability to undertake world-leading computational science research. DEISA has already proved its relevance for advancing computational sciences in leading scientific and industrial disciplines within Europe and has paved the way towards the deployment of a cooperative European HPC ecosystem. The existing infrastructure is based on the tight coupling of eleven leading national supercomputing centres, see figure 1, using dedicated network interconnections of GEANT2 (2008) and the National Research and Education Networks (NRENs).

Launched in 2005, the DEISA Extreme Computing Initiative (DECI, [3]) regularly selects leading grand challenge HPC projects, based on a peer review system and approved by the DEISA Executive Committee (Execom), to enhance DEISA’s impact on the advancement of computational sciences. By selecting the most appropriate supercomputer architectures for each project, DEISA is opening up the currently most
powerful HPC architectures available in Europe for the most challenging projects. This service provisioning model has now been extended from single project support to supporting Virtual European Communities. Collaborative activities will also be carried out with new European and other international initiatives. Of strategic importance is the cooperation with the Partnership for Advanced Computing in Europe (PRACE, 2008) which is preparing for the installation of a limited number of leadership-class Tier-0 supercomputers in Europe.

Emphasis will be put on contacts to research infrastructure projects established by the European Strategy Forum on Research Infrastructures (ESFRI, [8]), and the European HPC and Grid projects such as PRACE (2008) and EGEE (2008), respectively. The activity reinforces the relations to other European HPC centres, leading international HPC centres in Australia, China, Japan, Russia and the United States, and leading HPC projects worldwide, such as TeraGrid (2008) and NAREGI (2008). For supporting international science communities traversing existing political boundaries, DEISA2 will participate (e.g. in the Open Grid Forum, OGF, 2008) in the evaluation and implementation of standards for interoperation.

1. Vision, Mission, and Objectives

Vision: DEISA2 aims at delivering a turnkey operational solution for a future persistent European HPC ecosystem, as suggested by ESFRI [8]. The ecosystem integrates national Tier-1 centres and the new Tier-0 centres.

Mission: In DEISA2, the following two-fold strategy is applied: 1. Consolidation of the existing infrastructure developed in DEISA1 by guaranteeing the continuity of those activities and services that currently contribute to the effective support of world-
leading computational science in Europe. 2. Evolution of this infrastructure towards a robust and persistent European HPC ecosystem, by enhancing the existing services, by deploying new services including support for European Virtual Communities, and by cooperating and collaborating with new European initiatives, especially PRACE that will enable shared European PetaFlop/s supercomputer systems.

The objectives of the DEISA1 running from 2004 to 2008 running project were:
- Enabling terascale science by integrating Europe’s most powerful HPC systems. DEISA is a European supercomputing service built on top of existing national HPC services. This service is based on the deployment and operation of a persistent, production quality, distributed supercomputing environment with continental scope.
- The criterion for success: Enabling scientific discovery across a broad spectrum of science and technology. The integration of national facilities and services, together with innovative operational models, is expected to add substantial value to existing infrastructures.

The objectives of the current DEISA2 project running from the years 2008 to 2011 are:
- Enhancing the existing distributed European HPC environment (built in DEISA1) towards a turnkey operational infrastructure.
- Enhancing service provision by offering a manageable variety of options of interaction with computational resources. Integration of European Tier-1 and Tier-0 centres.
• The petascale Tier-0 systems need transparent access from and into the national data repositories.

2. The DEISA Infrastructure and Services

The Distributed European Infrastructure for Supercomputing Application is operated on top of national HPC services. It includes the most powerful supercomputers in Europe with an aggregated peak performance of about 1.3 PetaFlop/s in mid 2008 which are interconnected with a trusted, dedicated 10 Gbit/s network, based on GEANT2 (GEANT2, 2008) and the National Research and Education Networks (NRENs). The essential services to operate the infrastructure and support its efficient usage are organized in three Service Activities:

Operations Services refer to operating the infrastructure including all existing services, adopting approved new services from the Technologies activity, advancing the operation of the DEISA HPC infrastructure to a turnkey solution for the future European HPC ecosystem by improving the operational model and integrating new sites.

Technologies Services cover monitoring of technologies, identifying and selecting technologies of relevance for the project, evaluating technologies for pre-production deployment, planning and designing specific sub-infrastructures to upgrade existing services or deliver new ones based on approved technologies. The middleware components and services are described in more detail in chapter 4, see also (Niederberger, 2007).

Applications Services cover the areas applications enabling and extreme computing projects, environment and user related application support, and benchmarking. Applications enabling focuses on enhancing scientific applications from the DEISA Extreme Computing Initiative (DECI, [3]), Virtual Communities and EU projects. Environment and user related application support addresses the maintenance and improvement of the DEISA application environment and interfaces, and DEISA-wide user support in the applications area. Benchmarking refers to the provision and maintenance of a European Benchmark Suite for supercomputers.

3. DEISA Grid middleware

DEISA has a special service activity to deploy and operate generic Grid Middleware needed for the operation of the DEISA supercomputing Grid infrastructure. The services provided include “basic services”, which enable local or extended batch schedulers and other cluster features to simplify user access to the DEISA infrastructure. These basic services are enhanced by advanced services which allow resource and network monitoring as well as information services and global management of the distributed resources. Examples for these services are harmonization of national job management strategies, deployment, test und update of middleware like UNICORE, [35], [36], and Globus. Though most of these services are standard services in supercomputer environments, they have to be adapted to European distributed infrastructures.
User transparency is a necessity, i.e. users should not be bothered with the underlying grid technologies. The same holds for applications which are part of the corporate wealth of research organizations, i.e. only minimal intrusion on applications should be required, and applications should not be tied to a specific IT infrastructure. For this reason the UNICORE software [35], [36] is used as the middleware in the DEISA infrastructure to access the heterogeneous set of computing resources and managing workflow applications. Furthermore, in order to achieve interoperability with other leading Grid infrastructures in the world, new middleware is being evaluated, to decide about the best possible usage in the deployment of the infrastructure. Only fully production quality middleware (with RAS features) will be integrated into the production environment. As the specifications of OGSA and related standards are likely to evolve also middleware interoperability needs to be ensured.

Table 1. The DEISA operations services matrix.

<table>
<thead>
<tr>
<th>DEISA Services</th>
<th>Core Services</th>
<th>Additional Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>NREN, configuration and monitoring</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>MC-GPFS, GridFTP data staging</td>
<td>OGSA-DAI, SRB</td>
</tr>
<tr>
<td>Compute</td>
<td>Local batch systems, Resource Monitoring, Information System</td>
<td>UNICORE</td>
</tr>
<tr>
<td>AAA</td>
<td>LDAP, PKI, Accounting system, single sign on</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>DCPE modules, INCA, user support, trouble ticket system</td>
<td>DESHL, Portals</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration of additional site (associate) partners</td>
<td></td>
</tr>
</tbody>
</table>

Nowadays, leading scientific applications analyze and produce large amounts of data. Some of these applications need the computational capacities offered by the DEISA Grid.

With GPFS, DEISA has a very efficient high performance global file system for global data management and community access to data repositories, well adapted to High Performance Computing. It is based on the IBM Global Parallel File System (GPFS, 2008). This DEISA-wide shared file system enables the users to access their data transparently from every partner site. However, this technology does not cover all the global data management requirements. First of all, not all computing systems in DEISA can be integrated into the existing Global File Systems. Moreover, because of limited space on the DEISA global file systems, large amounts of data cannot be stored for an infinitely long time, and as a consequence data can not always be directly accessible from the applications running on the DEISA facilities. Before processing data they have to be transferred to a DEISA global file system or a local scratch system. Also, at the end of an application run, output data may have to be transferred to other storage facilities e.g. mass storage facilities of DEISA partners. Therefore DEISA has deployed a second high performance file transfer service based on striped GridFTP, which is also capable of taking advantage of the full network bandwidth for individual transfers. Last but not least, global file systems allow different applications to share the same data, but the opposite service is also needed: an application that needs to access a distributed dataset. Therefore the DEISA global data management roadmap focuses on the complementary objective of providing high performance access to distributed data sets, by enabling database management software like OGSA-DAI (OGSA-DAI, 2008)
or grid storage software like SRB (SRB, 2008). Moreover Grid based data transfers and Grid enabled data registration systems will provide DEISA users with facilities to retrieve and store data in Hierarchical Storage Management (HSM) facilities at DEISA sites and to register files independent of their physical location having global file names translated through registries and catalogues. Additionally it is planned to provide an uniform grid enabled access to specialized, in-house developed, or legacy databases by Grid enabled database access services independent of locations and formats of the databases. DEISA definitely will expand its data management capabilities in order to stay attractive for “grand challenge” applications.

The DEISA grid infrastructure permits a diversity of security policies. Within this virtual organization, users need transparent access to the DEISA Grid infrastructure with single sign-on facilities. Vice versa, partners need control on usage of their resources. These facilities, commonly referred to as Authentication, Authorization and Accounting services, must be trusted by all sites to protect their local environment against unauthorized access. Because of non direct contacts between users and remote DEISA sites dispatch services, a global administration had to be developed. Within DEISA a user only needs to contact a local administrator to get a DEISA POSIX (uid/gid) account. The user information will be stored into an LDAP services database which allows to update local information consecutively every day on all DEISA systems in a secure manner.

A secure single sign-on is realized via X.509 certificates [37] for authentication and authorization. DEISA trusts the certificates issued by the Certificate Authorities (CAs) accredited by the EuGridPMA [9], one of the members of the IGTF, a worldwide federation of CAs. This guarantees uniqueness of the certificates. Matching of uids and X.509 certificates (2008) allows the deployed Grid middleware to decide which services may be accessed. Because of the availability of the LDAP-information in all locations an XML-based database has been established which holds and presents all the relevant information for accounting. Aggregated reports will be created on resource usage by individual users and projects on a monthly basis.

The security of the DEISA infrastructure depends on the trust provided by the installed middleware that operates the services and on the security services that are used by the middleware as well as by the dedicated nature of the DEISA 10 Gb/s network infrastructure. Security issues related to networking below ISO/OSI layer 5, which is transport, network, link and physical layers, are very low, because of the dedicated nature of the network. The switching of connections or the insertion of packages into existing streams can be done by already known individuals residing on DEISA hosts, assuming no DEISA system has been hacked. Nevertheless an insider threat attack could be started. Because of this the Computer Emergency Response Team (CERT) teams of all organizations have to work closely together and have to exchange any kind of security incidences as soon as possible. A mutual cooperative trustfulness concerning vulnerability assessment will be indispensable.

The Applications and User Support Service Activity is in charge of all actions that enable or enhance the access to the DEISA supercomputing resources and their impact on computational sciences. It provides direct support to the major scientific initiatives of DEISA and helps users to run new challenging scientific applications. Large, demanding applications are running in parallel on several hundreds or thousands of processors in one specific site. Multi-site Grid applications are running concurrently on several systems, so that each component is executed on the most appropriate platform. Other applications are running at one site using data sets distributed over the whole
infrastructure. And finally, multiple applications are running at several sites sharing common data repositories. Also portals and Web interfaces used to hide complex environments from end users and to facilitate the access to a supercomputing environment for non-traditional user communities have to be supported.

To achieve these objectives, several activities have been deployed. The DEISA Common Production Environment (DCPE, 2008) is running on all platforms of the infrastructure, with a very high coherency across the homogeneous super-clusters and a weaker one across heterogeneous platforms. DCPE has been designed as a combination of software components (shells, compilers, libraries, tools and applications) available on each site and an interface to access these in a uniform way, despite the local differences in installation and location. DCPE is automatically monitored checking its behavior continuously and identifying unexpected problems. User support also includes documentations on access and usage of the DEISA supercomputing environment as well as installing a decentralized Help Desk. Last but not least training sessions are organized to enable fast development of user skills and know-how for the efficient utilization of the DEISA infrastructure.

4. Users and applications on the DEISA Infrastructure

Many grid initiatives aim at building a general purpose grid infrastructure and therefore have to cope with many barriers such as complexity, resource sharing, crossing administrative domains, handling IP and legal issues, dealing with sensitive data, interoperability, and facing the issue to expose every little detail of the underlying infrastructure services to the grid application user. DEISA is different from these grid initiatives in that it avoids most of these barriers by staying very focused: The main focus of DEISA is to provide the European supercomputer user with a flexible, dynamic, user-friendly supercomputing ecosystem for easy handling, submitting, and monitoring long-running jobs on the best-suited supercomputer(s) in Europe, trying to avoid the just mentioned barriers. In addition, DEISA offers application-enabling support.

For a similar European funded initiative specifically focusing on enterprise applications, we refer the reader to the BEinGRID project [2], which consists of 18 so-called business experiments each dealing with a pilot application that addresses a concrete business case, and is represented by an end-user, a service provider, and a Grid service integrator. Experiments come from business sectors such as multimedia, financial, engineering, chemistry, gaming, environmental science, and logistics and so on, based on different Grid middleware solutions, see (BEinGRID, [2]).

DEISA has another focus: capability computing with grand challenge scientific applications. Such large-scale simulations with highly parallel applications require the simultaneous usage of hundreds or thousands of tightly coupled processor-cores with a high bandwidth, low latency interconnect. Disciplines with such supercomputing needs include astrosiences/ cosmology, climate research, fusion energy research, bio sciences, materials and nanoscience, engineering, and others. Applications from all these areas were successfully used in DEISA by users from about 160 different European universities and research centres, with collaborators from four other continents (North and South America, Asia and Australia). The main DEISA HPC user requirements are:
Remote, standard, easy access to resources, applications, and data.

HPC users are usually conservative and have no interest in handling the complex middleware stacks.

Global login. HPC users want a single “European” username and uniform access.

Comfortable data access. HPC users want global, fast and comfortable access to their data, across all the DEISA HPC centres.

Common production environment. There is no need for an identical, but for an equivalent HPC software stack.

Global help desk. From their local HPC site or from any other DEISA site, users can access the central point of support contact.

Application support. HPC users need help in scalability, performance and adaptation of their application codes to different architectures.

These HPC user requirements are leading to the following DEISA HPC services:

- Running large parallel applications in individual sites by orchestrating the global workload, or by job migration services.
- Enabling workflow applications with UNICORE, enabling coupled multiphysics Grid applications.
• Providing a global data management service whose main objectives are the integration of distributed data with distributed computing platforms, and enabling efficient, high performance access to remote data with Global File System and striped GridFTP.

• Integrating hierarchical storage mgmt and databases in the Grid.

• Deploying portals to hide complex environments to new users, and to interoperate with other existing grid infrastructures.

The DEISA infrastructure essentially supports large single site capability computing through highly parallel batch jobs. Best suited, and, when required, most powerful supercomputer architectures are selected for each project. DEISA also supports multi-site supercomputing for many independent supercomputer jobs (e.g. for parameter scans) through various technical means (UNICORE [35], [36], DESHL [6], Multi-Cluster Loadleveler LL-MC, 2008, Application Hosting Interface, etc.) which greatly benefit from the DEISA global file system with a single name space, and from the GridFTP data management service.

DEISA supports mainly four kinds of applications: single job parallel programs for efficient usage of thousands of processor-cores (including parameter jobs, i.e. running one application with many different input parameters), data intensive applications with distributed file system support, workflow applications to combine several tasks (simulation, pre- and post-processing steps), and coupled applications. In the following, we describe application profiles and use cases that are well-suited for the DEISA supercomputing Grid, and that can benefit from the computational resources made available by the DECI Extreme Computing Initiative, [3].

International collaborations involving scientific teams that access the nodes of an AIX super-cluster in different countries, can benefit from a common data repository and a unique, integrated programming and production environment. Imagine, for example, that team A in France and team B in Germany dispose of allocated resources at IDRIS in Paris and FZJ in Juelich, respectively. They can benefit from a shared directory in the distributed super-cluster, and for all practical purposes it looks as if they were accessing a single supercomputer.

Workflow applications involving at least two different HPC platforms. Workflow applications are simulations where several independent codes act successively on a stream of data, the output of one code being the input of the next one in the chain. Often, this chain of computations is more efficient if each code runs on the best-suited HPC platform (e.g. scalar, vector, or parallel supercomputers) where it develops the best performance. Support of these applications via UNICORE (see [35], [36]) which allows treating the whole simulation chain as a single job is one of the strengths of the DEISA Grid.

Coupled applications involving more than one platform. In some cases, it does make sense to spread a complex application over several computing platforms. This is the case of multi-physics, multi-scale application codes involving several computing modules each dealing with one particular physical phenomenon, and which only need to exchange a moderate amount of data in real time. DEISA has already developed a few applications of this kind, and is ready to consider new ones, providing substantial support to their development. This activity is more prospective, because systematic
production runs of coupled applications require a co-allocation service which is currently being implemented.

Finally, two Joint Research Activities (JRA) complement the portfolio of service activities. JRA1 (Integrated DEISA Development Environment) aims at an integrated environment for scientific application development, based on a software infrastructure for tools integration, which provides a common user interface across multiple computing platforms. JRA2 (Enhancing Scalability) aims at the enabling of supercomputer applications for the efficient exploitation of current and future supercomputers, to cope with a production infrastructure characterized by aggressive parallelism on heterogeneous HPC systems at European scale.

5. DECI – DEISA Extreme Computing Initiative for supercomputing applications

The DEISA Extreme Computing Initiative (DECI, [3]) was launched in May 2005 by the DEISA Consortium, as a way to enhance its impact on science and technology. The main purpose of this initiative is to enable a number of “grand challenge” applications in all areas of science and technology. These leading, ground-breaking applications must deal with complex, demanding and innovative simulations that would not be possible without the DEISA infrastructure, and which benefit from the exceptional resources provided by the Consortium. The DEISA applications are expected to have requirements that cannot be fulfilled by the national services alone.

In DEISA2, the activities oriented towards single projects (DECI) will be qualitatively extended towards persistent support of Virtual Science Communities. This extended initiative will benefit from and build on the experiences of the DEISA scientific Joint Research Activities where selected computing needs of various scientific communities and a pilot industry partner were addressed. Examples of structured science communities with which close relationships have been or are to be established are the European fusion and the European climate communities. DEISA2 will provide a computational platform for them, offering integration via distributed services and web applications, as well as managing data repositories. The 2007 call of the DEISA Extreme Computing Initiative returned 63 submission, of which 45 were accepted by the DEISA Consortium. A few examples running on the DEISA infrastructure are:

- First-principles statistical mechanics approaches to surface physics and catalysis.
- Molecular switches at metal surfaces.
- Flame-driven deflagration-to-detonation transitions in supernovae.
- The role of plasma elongation on the linear damping of zonal flows.
- Turbulence driven by thermal gradients in magnetically confined plasmas.
- Interactions between neuronal fusion proteins explored by molecular dynamics.
- Coupled large-eddy simulations of turbulent combustion and radiative heat transfer.
- Frequency-dependent effects of the atlantic meridional overturning on the tropical pacific ocean.
Reynolds number effects on the Reynolds-stress budgets in turbulent channels.

- Dark galaxies and lost baryons.
- Quantitative accuracy Analysis in computational seismology.
- Effects of complicated 3D rupture geometries on earthquake ground motion and their implications.
- On the final spin from the coalescence of two black holes.
- Massively parallel quantum computer simulations: towards realistic systems.
- Role of protein frame and solvent for the redox properties of azurin from Pseudomonas aeruginosa.
- Metal adsorption on oxide polar ultrathin films.
- Ab-initio simulations of protein/surface interactions mediated by water.

6. The DEISA Long-Term Sustainability Strategy

Any project with a finite lifetime faces the challenge to successfully maintain and deploy the project results after the end of the project when funding runs dry. Therefore, during any project, a strategy has to be developed to ensure sustainability of the results after the project. This strategy contains several straight-forward requirements which usually have to be fulfilled and which should be verified during the project one by one. Some of the major requirements and activities to prepare for sustainability are:

- Towards the end of the project, results should be in a mature state such that users are easily able to accept and handle them. This can be demonstrated during the project’s second half through use cases or best practices.
- There will be a strong market or user demand for the results developed during the project. This can be demonstrated through market or user studies during the project.
- It has to be ensured that there will be enough resources available (financial, experts, equipment, support from stakeholders, etc) for a smooth transition of the results into the next phase beyond the end of the project.
- To guarantee widest visibility for the new results (and the organisation providing them) a dissemination and exploitation plan has to be developed during the project which is based on realistic facts and figures from trusted sources.
- Project management has to identify, eliminate or (at least) reduce potential barriers in all the different areas such as technology, culture, economics, and politics, and mental and legal barriers.

Only if each individual components withstand a sustainability test this will result in the sustainability of the ensemble.

As a demonstration example, we have checked these requirements for the DEISA project, especially for the technology, infrastructure, operations, services, expertise, communities, collaborations, economical, and political landscape (Gentzsch, 2008b). One of the main goals of DEISA2 is to ensure long-term sustainability of its main results and achievements:
• an operational, distributed European HPC infrastructure ready for use in the future European HPC ecosystem,
• well established European expert teams able to provide the necessary services, support for grand challenge computational science projects
• European user communities benefiting from a European HPC infrastructure

To reduce or eliminate existing technological, cultural, economic and political barriers and foster wider acceptance of the European HPC infrastructure, the overall DEISA2 sustainability model is based on ensuring sustainability of individual areas such as technology, infrastructure, operations, expertise, communities, collaborations, and the eco-political landscape. For these areas, the DEISA Consortium has identified the following assumptions and a wide variety of actions and measures:

Technology and Infrastructure:

• The DEISA infrastructure is built on existing, proven, sustainable technology components, including: the GÉANT2 and NRENs based high performance network; access to all major types of state-of-the-art supercomputers, based on national supercomputer services; a homogenized global software environment over the heterogeneous HPC architectures.
• DEISA2 is continuing, consolidating and extending the pioneering work of the former DEISA project, to deliver and operate a European supercomputing infrastructure and related services, ready for use in a European HPC ecosystem.

Operations and Services:

• DEISA2 infrastructure operations will benefit from the many-years operations of the individual European supercomputers centres. These individual operations will be orchestrated by the partners after the end of the funded project.
• For the effective support of computational sciences in the HPC area, activities relevant for applications enabling, operation, and technologies have been developed, and are now continued and further enhanced.

Expertise:

• DEISA stimulated tight collaboration of the expert groups in the different HPC centres: existing and developing expertise within the individual centres has been united through the current DEISA project and will be provided in the future to the wider European HPC communities.

Communities:

• The annual DEISA Extreme Computing Initiative (DECI,[3]), launched in 2005, supports the most challenging supercomputing projects in Europe which require the special resources and skills of DEISA (DEISA, [4]).
• The service provisioning model is currently being extended from one that supports single projects to one supporting Virtual European Communities.
• For supporting international science communities across existing political boundaries, DEISA2 participates in the evaluation and implementation of standards for interoperation.

Collaborations:

• Collaborative activities will be carried out with new European and other international initiatives.
• Emphasis will be put on contacts to research infrastructure projects established by the ESFRI, and the European HPC and Grid projects such as PRACE and EGEE, respectively.
• The activity reinforces the relations to other European HPC centres, leading international HPC centres and initiatives in Australia, China, Japan, Russia and the United States, and leading HPC projects worldwide, such as TeraGrid and NAREGI.

Eco-political landscape:

• The DEISA Consortium has been contributing to a raising awareness of the need for a persistent European HPC infrastructure as recommended by the European Strategy Forum on Research Infrastructures (ESFRI) in its report (ESFRI [8]).
• Of strategic importance is the cooperation with the PRACE project (PRACE 2008) which is preparing for the installation of a limited number of leadership-class Tier-0 supercomputers in Europe.

Short- to mid-term, DEISA has developed a collaboration strategy to encourage European users to use its HPC infrastructure, and to encourage non-European users through DECI to jointly apply, together with their European research colleagues, for HPC resources and related services. Also, DEISA aims at interoperating with existing infrastructure projects around the world to drive interoperability of the different infrastructure services, enabling users world wide to flexibly use the best suited resources for solving scientific grand challenge problems.

In summary, the DEISA services will have a good chance to be available still after the project funding dries out: because DEISA has a very targeted focus on specific (long-running) supercomputing applications and most of the applications just run on one – best-suited - system; because of its user-friendly access - through technology like DESHL [6] and UNICORE ([35], [36]); because of staying away from more ambitious general-purpose Grid efforts; because of its coordinating function which leaves the consortium partners (the European supercomputer centres) fully independent; and because of ATASKF (DECI, [3]), the application task force’ application experts who help the users with porting their applications to the DEISA infrastructure. If all this is here to stay, and the (currently funded) activities will be taken over by the individual
supercomputer centres, DEISA services will have a good chance to exist for a long time. And then, we might end up with a DEISA Cloud which will become an (external) HPC node within your grand challenge science application workflow.

An article available in the OGF Thought Leadership Series (Gentzsch, 2008a) offers 10 rules for building a sustainable grid infrastructure, which are mainly non-technical, because we believe most of the challenges in building and operating a grid are in the form of cultural, legal and regulatory barriers. These rules are derived from mainly four sources: research on major grid projects published in a RENCI report (Gentzsch, 2007a), the e-IRG Workshop on “A Sustainable Grid Infrastructure for Europe” (Gentzsch, 2007b), the 2nd International Workshop on Campus and Community Grids at OGF20 in Manchester (McGinnis, 2007), and personal experience with coordinating the German D-Grid Initiative (Neuroth, 2007, D-Grid, [7]).

7. Lessons Learned

Four years of DEISA production have shown that the concept implemented in DEISA has well succeeded. DEISA aimed at deploying a persistent basic European infrastructure for general purpose high performance computing, and now adapts to new FP7 strategies. This does not preclude that the organizational structure of DEISA may change because of merging with new HPC initiatives. But the general idea of DEISA will be sustained. One of the next challenges will be to establish an efficient organization embracing all relevant HPC organizations in Europe. Being a central player within European HPC initiatives, DEISA intends to contribute to a global e-Infrastructure for science and technology. Integrating leading supercomputing platforms with Grid technologies and reinforcing capability with shared petascale systems is needed to open the way to new research dimensions in Europe.

The operation of an infrastructure like DEISA leads to new management challenges not seen before. Managing a supercomputer system or a number of locally installed cluster systems differs heavily from a European supercomputer infrastructure where staff members dealing with the same problem are thousands of miles away from each other. There is no short cut, going to the office next door, just checking if we agree on some option settings within a software component. Within a virtual organization every small modification has to be checked by all partners over and over again. Installing new software components requires synchronization with all participants, if any dependencies exist. Scheduling of tasks, installations, system power up and down, network infrastructure changes and others have to be agreed on. Often, performing a task takes longer than estimated.

Our experience shows that many of these tasks cannot be handled only by e-mail. It is mandatory to have regular video or phone conferences, writing minutes and checking for completion of tasks. Additionally it is often necessary to agreed on strict rules for processing, especially in case of disagreements. Often, those dissents are found in security policy issues, scheduling of software installation and upgrades, and budget issues in the context of necessary components.

For this purpose the DEISA operation team has been established: planning and coordination of tasks, forwarding of information, power of decision, and managing in general are prerequisites for a successful European production quality infrastructure. Establishing this team has extremely simplified the collaborative work, and it should be
recommended to anyone dealing with similar infrastructures to start with adequate organizational and management structures.

Acknowledgement

This work is supported by the European DEISA project, funded through European Commission contracts RI-508830, RI-031513 and RI-222919. I want to thank the DEISA Team, and especially Stefan Heinzel, Hermann Lederer, Markus Ramp, Johannes Reetz, and Andreas Schott from the Max-Planck-Institute of Plasmaphysics in Garching, for their continuous support.

References

UNICORE 6 – A European Grid Technology

Achim STREIT, Sandra BERGMANN, Rebecca BREU, Jason DAIVANDY, Bastian DEMUTH, André GIESLER, Björn HAGEMEIER, Sonja HOLL, Valentina HUBER, Daniel MALLMANN, Ahmed Shiraz MEMON, Mohammad Shahbaz MEMON, Roger MENDAY, Michael RAMBADT, Morris RIEDEL, Mathilde ROMBERG, Bernd SCHULLER, Thomas LIPPERT

Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich GmbH
Germany

Abstract. This paper is about UNICORE, a European Grid Technology with more than 10 years of history. Originating from the Supercomputing domain, the latest version UNICORE 6 has matured into a general-purpose Grid technology that follows established Grid and Web services standards and offers a rich set of features to its users. An architectural insight into UNICORE is given, highlighting the workflow features as well as the different client options. The paper closes with a set of example use cases and e-infrastructures where the UNICORE technology is used today.

Keywords. Grid, Supercomputing, UNICORE, middleware, WS-RF, OGSA, e-infrastructure, DEISA, D-Grid, D-MON, OMII-Europe, Chemomentum

Introduction

Grid computing is a key technology that supports scientists and engineers both in academia and industry to solve challenging problems, to enhance their productivity when working with complex environments, and to collaborate in unprecedented ways. Grids integrate distributed computing resources – from supercomputers, mid-range commodity clusters to desktop computers –, data produced by scientific instruments – such as tomographs, accelerators, satellites, or telescopes –, data created through simulations and stored in archives, repositories, and databases, and visualisation media through high-speed networks. Most importantly, Grids should also integrate the users who are accessing this infrastructure.

The resulting knowledge and working environment, also called e-infrastructure [1], allows members of virtual organisations, scientific communities and individuals to pursue research and development with increased efficiency as well as accessing and managing data in novel ways. Grid technologies integrate different computing models like synchronous high-performance computing, high-throughput computing, data-intensive computing, on-demand computing, and collaborative computing through a
common set of protocols and interfaces. Thus, it saves scientists and engineers from having to cope with multiple, incompatible and inconsistent environments and technologies while working with different kinds of resources. Grids form the basis for e-Science (electronic or enhanced Science) [2].

In the last two years activities in Grid computing have changed; in particular in Europe the focus moved from pure research-oriented work on concepts, architectures, interfaces, and protocols towards activities driven by the usage of Grid technologies in day-to-day operation of e-infrastructure and in application-driven use cases. This change is also reflected in the UNICORE activities [3]. The basic components and services have been established, and now the focus is increasingly on enhancement with higher level services, integration of upcoming standards, deployment in e-infrastructures, setup of interoperability use cases and integration of applications.

The development started back more than 10 years ago, when in 1996 users, supercomputer centres and vendors were discussing “what prevents the efficient use of distributed supercomputers?”. The result of this discussion was a consensus which still guides UNICORE today; seamless, secure and intuitive access to distributed resources. Consequently a project proposal for the German Ministry of Education and Research (BMBF) was submitted and the first UNICORE project started on the 1st of August 1997. In this project the foundation of the first UNICORE versions were initiated including a well defined security architecture based on X.509 certificates, an intuitive graphical user interface programmed in Java, and a central job supervisor component. The successful end of the first project in December 1999 [4], resulted in follow-up project UNICORE Plus from January 2000 until December 2002 [5]. Here the focus was on implementation enhancements, the replacement of Codine by a custom job supervisor components called the NJS, an integration of extended job control features for workflows, and the implementation of several application specific user interfaces [6].

Since the end of 2002 continuous development of UNICORE took place in several EU-funded projects, with the subsequent broadening of the UNICORE community to participants from across Europe. In 2004 the UNICORE software became open source and since then UNICORE is developed within the open source developer community. Publishing UNICORE as open source under BSD license has promoted a major uptake in the community with contributions from multiple organisations. Today the developer community includes developers from Germany, Poland, Italy, UK, Russia and other countries under the leadership of JSC.

The structure of the paper is as follows. In Section 2 the driving forces and design principles for UNICORE 6 are presented. Section 3 gives an insight on the architecture and technical details of UNICORE, while Section 4 covers the broad range of UNICORE clients. In Section 5 several use cases and e-infrastructures are described where the UNICORE technology is used. The paper closes with a conclusion.

1. Driving Forces and Design Principles

UNICORE has a background in supercomputing, following the principle “Grid driving Supercomputing”, and thus the UNICORE developer community maintains a close feedback loop with major stakeholders in the supercomputing domain. With supercomputer users being the end customers using in particular the various existing UNICORE clients, their needs and requirements are closely monitored and catered for
in the development cycle. This is done through mailing lists, feature tracking tools, regular meetings and training sessions at major supercomputing centres or infrastructures, as well as through personal communication channels. Prominent examples emerging from this process is the new UNICORE command-line client and DESHL (cf. Section 4).

Complementing end users, close feedback loops are maintained with supercomputing user support teams and operations staff. User support teams typically have a broader view on the user community in a supercomputing centre and are able to provide more generalised input to the development cycle. This is due to the fact that the user support team is often the first level support contact for the average supercomputing end user. For example at JSC a close collaboration with the user support is maintained through regular meetings discussing new features, requirements and capabilities. In addition, maintaining good contact to the operations teams is important. As they are tasked with deploying and running the UNICORE backend services on a 24-7 basis, it is important to receive their feedback to improve the installation and configuration process of the UNICORE services. This is done to establish a smooth and seamless integration of UNICORE into the operational model of a supercomputing centre.

Running UNICORE in a supercomputing centre should not require fundamental changes to any single operational procedure already in place, for example which operating systems to use, how to manage users or how to do the accounting. This seamless integration into existing environments has a strong influence on the design principles of the UNICORE technology in general.

Consequently, UNICORE is the European Grid technology for supercomputing e-infrastructures in Europe. UNICORE is used in DEISA [7] (cf. Section 5), the Distributed European Infrastructure or Supercomputing Applications, which couples the most powerful supercomputing systems in Europe and thereby operates and enhances a European supercomputing service on top of these existing national services. Through deploying and operating a set of core services and providing technologies (i.e. UNICORE) and excellent support to facilitate supercomputing applications, DEISA delivers a turnkey operational solution for the future persistent European Supercomputing ecosystem.

In PRACE [8], the Partnership for Advanced Computing in Europe, which prepares the deployment of a future pan-European PetaFlop supercomputing service, UNICORE is a prime candidate to be used to interconnect Europe’s future leadership supercomputers in combination with other core service operated in DEISA. The emerging ecosystem will also integrate European and national Grid initiatives, where UNICORE 6 establishes standards-based interoperability with other Grid technologies.

On the national level, UNICORE is used in the Gauss Centre for Supercomputing (GCS) [9], the alliance of the three German national supercomputing centers. These centers join their forces, resources and know-how to establish world-class supercomputing services to the German and European scientific community with leadership systems of the highest performance class.

Although being supercomputing-oriented, UNICORE is a general-purpose Grid technology. It can be used in Grid infrastructure of any nature and without limitations on the type of computing resources ranging from single PCs coupled together for a Campus Grid or cluster-systems similar to the EGEE Grid infrastructure. For example UNICORE is used in D-Grid [10] (cf. Section 5), the German national Grid initiative.
Based on the driving forces described above, several design and guiding principles for the new version UNICORE 6 were defined since 2004. Foremost to mention is the license policy of the UNICORE Grid technology. As the previous version UNICORE 5, UNICORE 6 is open source under BSD licence. Through using the open source platform SourceForge [11], contributions with own developments are easily possible. Further design principles are as follows:

- Standards-based: UNICORE 6 should be based on Grid and Web services being conform to the OGSA [12] and compliant with WS-RF [13]. It should implement the latest standards in various areas such as security, job management, data management, accounting, etc. of the Open Grid Forum [14], OASIS [15] and W3C [16].
- Open, extensible, interoperable: UNICORE 6 should be a modern Service-Oriented Architecture (SOA), which allows easy replacement of particular components with others. For example, it should be possible to plug-in different workflow components, which comply with domain-specific requirements. UNICORE 6 should be interoperable with other Grid technologies to enable a coupling of Grid infrastructures according to users needs.
- End-to-end, seamless, secure, and intuitive: UNICORE 6 should follow a vertical, end-to-end approach, offering components at all levels of a modern Grid architecture from intuitive user interfaces down to the resource level. Like previous versions UNICORE 6 should be seamlessly integrated into existing environments. In fact the keywords seamless, secure and intuitive were already coined at the beginning of UNICORE in 1996/7 (cf. Introduction).
- Mature security: UNICORE 6 should provide the security mechanisms adequate for the use in supercomputing environments and Grid infrastructures. X.509 certificates should form the basis for authentication and authorisation, enhanced with a support for proxy certificates and virtual organisations (VO) based access control.
- Workflow support: UNICORE 6 should comprise support for workflow jobs deeply integrated into the stack. At the same time this support should be extensible in order to use different workflow languages and engines.
- Application integration: Providing concepts to support applications in general is one of the turn key capabilities of Grid technology to convince users of the benefits from using the Grid. Hence, UNICORE 6 should provide well-designed mechanisms on the client, services and resource level for a tight integration of various types of applications from the scientific and industrial domain.
- Variety of clients: UNICORE 6 should come with different clients serving the needs of various scientific communities. For example, bio-life-scientists are typically used to work with graphical clients in order to define their complex workflows, while physicists are used to work with command-line tools. Hence, UNICORE 6 should offer graphical, command-line as well as portal/Web-based clients.
- Quick and simple installation and configuration: To address requirements from operational teams and to lower the barrier of adopting Grid technologies, the installation of UNICORE 6 should be straight-forward and quick.
Similarly the configuration of the various services and components should be easy to handle without too many cross-references. For example, it should be possible to configure and test one component after the other and thereby ramping up the operational status of a UNICORE 6 installation incrementally.

- Support for many operating and batch systems: Driven by the supercomputing domain, UNICORE 6 should be working on various kinds of operating systems without being tied to a specific operating system. Naturally this applies to the various clients, but in particular for the server-side components as well. Furthermore, different batch systems have to be supported, as different supercomputers have different batch systems like LoadLeveler for IBM systems, NQE/NQS for NEC systems, or Torque as an open source software.

Finally, UNICORE 6 - clients and all service components - should be implemented in a platform-independent way.

### 2. Architecture and Standards

The architecture of UNICORE 6 is three-layered in client layer, service layer and system layer as shown in Figure 1.

#### 2.1. Client Layer

On the top layer a variety of clients are available to the users, ranging from a programming API named HiLA (cf. Section 3.3), graphical clients such as the Eclipse-based URC (UNICORE Rich Client, cf. Section 3.1) to a command-line interface named UCC (UNICORE Command-line Client, cf. Section 3.2). For more details on these clients see Section 3. For a tight integration of various types of applications, the concept of GridBeans [20] was invented, which offers an API to easily implemented graphical client extensions and connect them with UNICORE 6’s core functionalities. Complementing these, UNICORE 6 services can be accessed through portal or Web-based technologies, e.g. GridSphere [44] or UNICORE/w3 [17], which are available as beta software.

#### 2.2. Service Layer

The middle layer comprises all services and components of the UNICORE Service-Oriented Architecture (SOA). The Gateway component [18] acts as the entry point for a UNICORE site and performs the authentication of all incoming requests. The XNJS component [19] is the job management and execution engine of UNICORE 6. It performs the job incarnation, namely the mapping of the abstract job description to the concrete job description for a specific resource. The functionality of the XNJS is accessible via two service interfaces in UNICORE’s WS-RF hosting environment. UNICORE’s proprietary interface is called UAS (= UNICORE Atomic Services) [20] and offers the full functionality to higher level services, clients and users. In addition to the UAS, a standardised set of interfaces based on open, common standards is available in UNICORE 6 (depicted as “OGSA-*” in Figure 1, cf. Section 2.4).
For authorisation of users the XNJS uses the XUUDB user database to perform the mapping from X.509 certificates to the actual users’ logins. The XUUDB component is a Web service in itself, which allows it to be used from multiple UNICORE installations, e.g. within the same computing centre.

Like in many service-oriented environments, a service registry is available, where the different services can register once they are started. A single service registry is necessary to build-up and operate a distributed UNICORE infrastructure. This service registry is contacted by the clients in order to “connect to the Grid”.

From the beginning, workflow support is of major importance for UNICORE. The two layered design with a separation of the workflow engine and the service orchestrator was primarily done for better scalability, but also offers the possibility to plug-in domain-specific workflow languages and workflow engines. In the EU-project Chemomentum a Shark open-source XPDL workflow engine was implemented. This workflow engine is shipped with UNICORE, as it provides all requirements for UNICORE’s workflow functionalities. Besides simple job-chains, loops, workflow variables and conditional execution are supported. The service orchestrator deals with brokering the execution and monitoring of the workflow and its respective parts and provides call-back mechanisms to the workflow engine. The resource brokering is performed via pluggable strategies in combination with the Resource Information Service. More details are found in [21]. The workflow capabilities are offered to the users on the client layer via the UNICORE client based on the Eclipse framework. Furthermore, the definition, submission, monitoring and control of workflows is also possible from the command-line client UCC. The Tracing Service collects runtime information from the workflow system, and allows generating performance metrics.
End users can use the tracer service to visualise the execution of a complex workflow from within the Eclipse-based client.

2.3. System Layer

On the bottom layer the TSI (Target System Interface) component is the interface between UNICORE and the individual resource management/batch system and operating system of the Grid resources. In the TSI component the abstracted commands from the Grid are translated to system-specific commands, e.g. in the case of job submission, the specific commands like *llsubmit* or *qsub* of the batch system are called. The TSI component is performing the proper setting of users’ UID and invocation of his/her environment. If a UNICORE installation should be operated with multiple users, the TSI component is the only component of the UNICORE 6 stack that needs to be executed with root privileges. All other UNICORE 6 components at a site can be executed under a standard user account, preferably a dedicated, UNICORE-related account. Note that the TSI component remained unchanged from UNICORE 5 to UNICORE 6. This has two major benefits. Firstly, the TSI is available for a variety of commonly used batch systems such as Torque, LoadLeveler, LSF, SLURM, OpenCCS, etc. Secondly, the migration of a UNICORE site from UNICORE 5 to UNICORE 6 is easier, as already used and well-tested TSI components can be retained, so that the adaptation of the TSI to a specific Grid resource with its system configuration and environment must not be repeated.

The USpace is UNICORE’s job directory. A separate directory exists for every job, where the XNJS and TSI stores all input data and where stdout and stderr are written to. For a site-to-site transfer and in particular for data transfer from/to external storages the GridFTP transfer protocol can be used.

2.4. Standards

Several standards from the Open Grid Forum and OASIS are used in UNICORE 6 in various domains (cf. to the boxes on the right in Figure 1). A full Web services stack based on WS-RF 1.2, SOAP, and WS-I is implemented to build UNICORE’s service-oriented architecture.

In security, full X.509 certificates are used as base line, while the access control is based on XACML policies. A support for SAML-based VOMS (virtual organisation management service) [22] was recently added as well as support for proxy certificates.

In the area of information systems, monitoring and accounting, the development of CIS, a GLUE 2.0 based information service for UNICORE 6 [23], is taking place in close collaboration with the GLUE working group in OGF. It gathers both static and dynamic information from all connected XNJS, which are then displayed either in raw XML or human-readable text form. As longitude and latitude information is also stored, a Google maps view allows a geographical representation of the Grid infrastructure. The OGSA-RUS interface for accounting in UNICORE 6 stores its data in the UR format [24].

In the area of job management, OGSA-BES and HPC-P are used for the creation, monitoring and control of jobs, whilst job definition is compliant with the JSDL (+ JSDL HPC extensions) standard [25]. On the system layer a TSI version for the DRMAA standard [26] is available enabling a standardized interface between the TSI and the batch system. In the area of data management and transfer, OGSA-ByteIO can
be used for data transfer, both for site-to-site and client-to-site transfers [27]. For a transfer of data from and to external storage, the GridFTP transfer protocol can be used.

3. Using UNICORE

In the following we will describe UNICORE’s core clients; namely the Eclipse-based graphical URC, the command-line client UCC as well as the programming API HiLA.

3.1. Eclipse-based URC

Since the beginning, UNICORE has offered graphical clients realising UNICORE’s baseline slogan by providing seamless, secure and intuitive access to Grid resources. Today the Eclipse-based client [28], shown in Figure 2, constitutes the most complete implementation of this idea.

Basing the major UNICORE client on the Eclipse rich client platform comes with several benefits. First of all, Eclipse is widely known and commonly used due to its well designed and flexible graphical interfaces. This lowers the entry barrier for new users, as many of them are already familiar with the tool.

Furthermore, although being written in Java, an Eclipse-based application contains some platform specific code. Through this approach, it looks just like a native application with a smoother integration into different platforms.

Finally, the Eclipse platform has a very sophisticated plugin mechanism: every software component in an Eclipse-based client is a plugin, each of which adding a well-defined range of functions. Plugins interact with each other in various ways and almost every plugin provides programming interfaces which can be used to extend its functionality and outer appearance. Following this paradigm, the graphical UNICORE client is extremely extensible. For instance, integration of new Grid services or

![Figure 2. UNICORE Rich Client (URC) showing the integration of the AMBER package [29]](image-url)
scientific applications is already accounted for in its design.

This client targets a wide range of users with varying Grid and IT experience. It provides a useful graphical view of the Grid, which can be filtered in order to find specific resources, services or files.

It can be used to submit computational jobs to Grid resources. To this end, small software packages provide tailored graphical user interfaces for many scientific applications available on the Grid. The same packages are responsible for visualising the output data of scientific simulations once the jobs have been executed and output files have been downloaded to the client machine.

Detailed resource requirements for jobs (e.g. required number of CPUs, amount of RAM) can be specified. Users are enabled to design complex scientific workflows that combine several applications to automate the completion of difficult tasks. To this end, a fully-fledged workflow editor is provided. It allows for graphical programming where building blocks like loops or if-statements can be arranged with just a few mouse clicks.

Security and access control are essential aspects in distributed computing. Dedicated panels deal with setting up security options, so users can specify whom they trust and how to identify themselves on the Grid.

Experienced users can perform various administrative tasks on the Grid by accessing specific parts of the user interface.

3.2. Command-line client UCC

UCC [30] is a very versatile command-line tool that allows users to access all features of the UNICORE service layer in a shell or scripting environment (cf. Figure 3 for an
overview of available commands). It allows to run jobs, monitor their status and retrieve generated output, both in single job mode or in a powerful and flexible batch mode for multiple jobs. Additionally, workflows can be submitted and controlled with the UCC.

UCC includes several data management functions. Remote storages can be listed and files can be transferred from local to remote as well as from server to server. UCC can be used for administrative purposes as well, for example to list all jobs, or to perform some clean up. An important feature of UCC is its extensibility. New commands can easily be added, and the “run-groovy” command allows the execution of scripts written in the Groovy programming language. A dedicated UCC mode for the popular Emacs editor is also available.

3.3. Programming API HiLA

HiLA is a High Level API for Grid Applications [31], that allows simple development of clients with just a few lines of code for otherwise complex functionality. It provides a single interface with multiple implementations for UNICORE 5, UNICORE 6 and OGSA-BES. HiLA was mainly developed in the EU funded DEISA [7] and A-WARE [32] projects. It is used to integrate UNICORE 6 access into DESHL [33] (including SAGA support), as part of the DEISA portal, and to connect GAT [34] with UNICORE 6. The nature of the HiLA API leads to a concise coding toolkit for building collective tier Grid services and client interfaces. Following the UNICORE principle of seamlessness, the design of the API models the Grid through a Object-oriented façade, presenting abstract representations of the underlying resources. Importantly, this includes encapsulating security configuration behind well defined interfaces, further enhancing the API.

The resources of the Grid are named following a URI naming scheme, for example, uncore6:/sites/FZJ_JUGGLE/storages/home or ogsa:/sites/GROW/
tasks/910c9b56-d497-46f8-960f-eaee43e1af37. The object navigation is based on a container/item model. An example source code is shown in Figure 4.

Navigation of locatable resources is done generically. Types of objects referenced by locations can be inferred and thus it is possible to call object specific operations such as size() or isDirectory() on Files. HiLA is currently evolving further to allow it to operate concurrently over multiple resources to perform powerful collective data management and monitoring.

```java
Location l = new Location("unicore6:/sites/GROW/tasks/910c9b56-d97-46af37");
Task t = HiLAFactory.getInstance().locate(l);
assertTrue(TaskStatus.RUNNING, t.status());
List<File> fl = t.getOutcomeFiles();
```

**Figure 4.** Example HiLA code
4. Use Cases

4.1. D-Grid

D-Grid is the German Grid infrastructure funded by the Federal Ministry of Education and Research (BMBF) [35]. It aims to provide reliable high-performance computing resources and related services for scientific communities in Germany. The first D-Grid projects started in 2005, the current projects will run until 2010, and plans for new projects continuing D-Grid after 2010 are underway.

Depending on the applications from the scientific communities different middleware solutions are supported in D-Grid, amongst them UNICORE, and many resources are accessible through more than one middleware [36]. Operating multiple middleware technologies in one Grid and, in many cases, even on the same resources raises various kinds of interoperability issues. To set up a comprehensive monitoring system spanning all the monitoring systems of the different middleware technologies and other available information providers, the D-MON project has been established [37]. Beside collecting and storing information, one of its main tasks is to unify the collected data and provide it to end users or other services. Here, the CIS (cf. Section 2 and [23]) is used to gather information from UNICORE 6 sites, and it is being adapted to read data back from the central monitoring system which it cannot provide itself.

4.2. DEISA

DEISA (Distributed European Infrastructure for Supercomputing Applications) [7] is the consortium of the 15 leading national supercomputing centres in Europe. They are tightly connected in the DEISA e-infrastructure, reaching about 1 PetaFlop/s of aggregated performance. Funded through the DEISA, eDEISA (both FP6 3) and DEISA2 (FP7) projects, DEISA operates and enhances a European supercomputing service on top of existing national services. This service is based on the deployment and operation of a persistent, production quality, distributed supercomputing environment with pan-European scope, which delivers a turnkey operational solution for the future persistent European HPC ecosystem.

A set of core services are operated in DEISA. This comprises a dedicated 10 Gbit/s network between all sites, a distributed parallel file system to enable remote I/O and data sharing, and a common production environment. On top of these core services, UNICORE is used as the Grid Middleware for workflow management and as an alternative access method to the supercomputers [36]. At present UNICORE 5 is still in production in DEISA, while the migration to UNICORE 6 is being prepared through an intensive pre-production phase with testing components and adapting configurations. DESHL – DEISA’s command-line client – is based on HiLA (cf. Section 3.3), resp. UNICORE 5. The SIMON (SIte MONitor) tool [38] was developed within DEISA to monitor the UNICORE 5 installations at the sites and to automatically inform the operations team of DEISA in case of problems or off-line components.

3 Framework Programme 6 (FP6) of the European Commission
4.3. OMII-Europe

The focus of the OMII-Europe project [39] funded under FP6 by the EU was on interoperability and usability of Grid infrastructures by providing standards-based Grid middleware leveraging existing work and standards. Targeted middleware platforms were the three major Grid technologies in the world: gLite, Globus and UNICORE. To achieve this, common Grid components for database access (OGSA-DAI), for virtual organisation management (VOMS), for portal solutions (GridSphere), as well as open standards such as OGSA-BES for job submission and OGSA-RUS for accounting were integrated and implemented in the three targeted middleware platforms. All these developments were put together in an integration activity in order to apply application use-cases [40], that require interoperability of Grid middleware and e-infrastructures to solve their grand challenge problems. This was enhanced by work towards a common security framework across all middleware technologies. UNICORE 6 was augmented with functionality that supports SAML-based VO management services like UVOS [41] or VOMS [42] from gLite.

4.4. Chemomentum

The Chemomentum project [21] uses UNICORE 6 to provide a Grid-based environment for defining and running complex workflows from bio-informatics. As a prime example, chemical properties are modelled using structural descriptors. Application areas include toxicity prediction, drug design and environmental risk.

![Figure 5. URC with Chemomentum application MOPAC](image-url)
assessment. Data management, i.e. storage, meta data handling and provenance tracking plays a major role in the project. Many chemical applications such as GAMESS, MOPAC and CODESSA have been integrated into this environment, and can be included in user-defined workflows (cf. Figure 5).

4.5. A-WARE

The solution developed in the A-WARE (An Easy Way to Access Grid Resources) project [32], funded under FP6, is a framework for building higher-level collective services focusing specifically on workflow. Particular care was taken to ensure that accessing the functionality can take place through multiple channels, in particular through a Web-based interface. This interface is based on two well known portal solutions, EnginFrame [43] and GridSphere [44] (respectively commercial and open-source). This provides the user with access to their resources in the Grid, including performing workflow over them. Implemented with an Enterprise Service bus, this middle-tier component hosts the collective layer workflow services and provides tools for managing each individual Grid resource. Thus it has a mediating role, acting as a conduit between the user and their resources.

A use-case in the fluid dynamics domain provided by Airbus involving multiple applications over a set of resources, and including data transfer was used to guide and validate the architecture. Besides the 'usual' Grid standards, the JBI (Java Business Integration) [45] and BPEL (Business Process Execution Language) [46] enterprise standards were used to beneficial effect in the project.

4.6. Phosphorus

The EU project Phosphorus [47] aims at coupling various research network test-beds in Europe demonstrating on demand service delivery across multi-domain/multi-vendor research network test-beds. This makes applications aware of their complete Grid resources (computational and networking) environment and capabilities, and able to make dynamic, adaptive and optimized use of heterogeneous network infrastructures connecting various high-end resources. Several use-cases are set up to demonstrate the integration between application, middleware and transport networks as well as to investigate and evaluate further technological development needs arising from the use-cases.

One such use-case is about collaborative data visualization using the KoDaVIS toolkit [48]. Besides integration KoDaVIS in UNICORE 6, a GridBean (cf. Figure 6) was developed to setup and control the collaborative visualization session with multiple users. This includes the reservation of bandwidth in the network as well as the monitoring of the performance of the collaborative visualization session.

4.7. Commercial usage at T-Systems

T-Systems SfR uses UNICORE within a commercial environment. T-Systems is operating the C²A²S²E-Cluster for its customers Airbus and DLR (the German Aerospace Centre) in cooperation with Airbus. The cluster is used to perform validated, industrialized simulation codes for application challenges from aircraft development, so “flying the virtual aircraft”. UNICORE is used here to provide access for DLR customers as well as foreseen third-party customers from outside DLR.
4.8. Usage in the NIC

The Jülich Supercomputing Centre (JSC) is part of the John von Neumann Centre for Computing (NIC) [49] and is responsible for operating several supercomputer systems. Through UNICORE users can access all computing resources operated within the NIC. The largest system is the IBM Blue Gene/P system with 65,536 processors and a peak performance of 223 TFlop/s [50]. As general purpose system for smaller jobs a IBM Power 6 575 based system with 448 processors and 8.4 TFlop/s peak performance is available [51]. About 450 users in 200 research projects are using these systems. A peer-review process of scientific proposal regulates the access to these systems. In addition several smaller clusters are operated, e.g. a 264 processor system for the soft matter physics community [52] and a 176 processor system for the D-Grid community [53].

5. Conclusions and Future Developments

In this paper we presented UNICORE, a European Grid Technology with more than 10 years of history. Starting from describing the roots of UNICORE in the German Supercomputing community in the mid 1990s, the latest version UNICORE 6 has matured into a general-purpose Grid technology that follows the latest Grid and Web services standards and offers a rich set of features to its users.
To achieve this several design principles and guidelines for UNICORE 6 were derived during 2004-2006. As a result UNICORE 6 has adopted the following guiding principles and implementation strategies: based on Grid and Web services standards, open and extensible SOA, interoperable with other Grid technologies, mature security mechanisms, seamless and intuitive, support for workflow, integration of applications, providing a variety of clients, quick and simple to install and configure, support for various operating and batch systems, implemented in Java.

The development is fostered by UNICORE’s licence policy; with being open source under BSD license a major uptake in the community was achieved and today multiple organisations and individuals are contributing the development of UNICORE. Through using the world’s largest Open Source software development Web site SourceForge, contributions are easily possible. In addition an environment for the developers is provided which contains mailing lists, trackers for bug reports and feature requests, a download page that links to all released UNICORE software components as well as a source code repository as the central storage of all UNICORE-related source code. This enables the community to grow and makes future development efforts open to the public.

Future versions of UNICORE 6 will remain up-to-date with latest research trends in the distributed systems and Grid domain (e.g. Web 2.0, REST, Grid license management, virtualisation, Cloud computing, Green-IT) and will continue to include new features resulting from user requirements (e.g. improved support for MPI execution environments and multi-core CPUs). Topics where additional functionality will be added or existing functionality will be enhanced are e.g. management of scientific data including scalable storage, metadata support, and data access, as well as administration, governance, and monitoring including improved service lifecycle management, performance metrics, resource usage control and billing.

Acknowledgement

The work summarized in this paper was done by many people. We gratefully thank them for their past, present, and future contributions in developing the UNICORE technology. Most of the work described here was supported and funded by the Helmholtz Programme “Scientific Computing” and various projects from the European Commission and BMBF under the respective contract numbers.

References

[41] UNICORE Virtual Organisation System (UVOS), http://uvos.chemomentum.org/
This page intentionally left blank
Chapter 4
Cloud Technologies
Cloud Computing for on-Demand Grid Resource Provisioning

Ignacio M. LLORENTE ², Rafael MORENO-VOZMEDIANO and Rubén S. MONTERO

Facultad de Informática, Universidad Complutense de Madrid
28040 Madrid, Spain

Abstract. The use of virtualization, along with an efficient virtual machine management, creates a new virtualization layer that isolates the service workload from the resource management. The integration of the cloud within the virtualization layer, can be used to support on-demand resource provisioning, providing elasticity in modern Internet-based services and applications, and allowing to adapt dynamically the service capacity to variable user demands. Cluster and grid computing environments are two examples of services which can obtain a great benefit from these technologies. Virtualization can be used to transform a distributed physical infrastructure into a flexible and elastic virtual infrastructure, separating resource provisioning from job execution management, and adapting dynamically the cluster or grid size to the users' computational demands. In particular, in this paper we analyze the deployment of a computing cluster on top of a virtualized infrastructure layer, which combines a local virtual machine manager (the OpenNebula engine) and a cloud resource provider (Amazon EC2). The solution is evaluated using the NAS Grid Benchmarks in terms of processing overhead due to virtualization, communication overhead due to the management of nodes across different geographic locations, and elasticity in the cluster processing capacity.

Keywords. Cloud computing, On-demand resource provisioning, Cluster and grid computing

Introduction

Recently, virtualization has brought about a new utility computing model called cloud computing, for the on-demand provision of virtualized resources as a service. Amazon Elastic Compute Cloud (Amazon EC2) [1], GoGrid [2] and FlexiScale [3] are examples of this new paradigm for elastic capacity provision. These systems are used as standalone resource providers, without any integration with the in-house infrastructure.

Our position is that this resource provision model can be seamlessly integrated with the in-house infrastructure when it is combined with a virtual machine (VM) manage-

---

¹This research was supported by Consejería de Educación de la Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through BIOGRIDNET Research Program S-0505/TIC/000101, by Ministerio de Educación y Ciencia, through the research grant TIN2006-02806, and by the European Union through the research grant RESERVOIR Contract Number 215605
²Corresponding Author: Full Professor, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain. E-mail: llorente@dacya.ucm.es
ment system. A VM manager is responsible for the efficient management of the virtual infrastructure as a whole, by providing basic functionality for the deployment, control and monitoring of VMs on a distributed pool of resources. Usually, these VM managers also offer high availability capabilities and scheduling policies for VM placement and physical resource selection.

One of the main goals of this work is to show the benefits of this architecture, which totally decouples the infrastructure management from the service management, and enables the dynamic provisioning of virtual resources in an on-demand basis to adapt the infrastructure to the service requirements. This approach is fully transparent for the service itself, and independent of the type of service. Furthermore, this provisioning model can be integrated with external cloud providers, to provide additional elastic capacity to the virtual infrastructure when the service demands increase or to satisfy peak demand periods.

The proposed architecture is illustrated with the OpenNebula VM manager [4,5], which provides the functionality needed to deploy, monitor and control VMs on a pool of physical resources. OpenNebula is an open source software, highly flexible and modular, which provides a centralized management of VMs and physical resources, support for adaptable scheduling policies and re-allocation policies for fault tolerance (high availability), and drivers for federation, which enable OpenNebula to interoperate with other remote OpenNebula engines or with clouds providers (such as Amazon EC2).

Cluster and grid communities are paying much attention to the evolution of clouds and virtualization technologies, since they can give support for deploying cluster and grid platforms on top of virtualized infrastructures, overcoming the main limitations of current physical platforms, such as resource heterogeneity, partitioning and isolation of running services and applications, node customization and configuration, etc. In this context, virtualization can be used to transform a distributed physical infrastructure into a flexible and elastic virtual infrastructure, separating the resource provisioning from the job execution management, and adapting dynamically the cluster or grid size to the users’ computational demands.

In order to evaluate the performance of the proposed architecture, we have deployed a computing cluster on top of a virtualized infrastructure layer, which combines a local OpenNebula manager and a cloud resource provider (Amazon EC2). The solution is evaluated using the NAS Grid Benchmarks [6,7] in terms of processing overhead due to virtualization, communication overhead due to the management of nodes across different geographic locations, and elasticity in the cluster processing capacity.

This work is organized as follows: Section 1 introduces the OpenNebula Virtual Infrastructure Engine; in Section 2 we analyze how cluster and grid platforms can be developed on top of virtualized architectures; in Section 3 we describe the proposed architecture for the elastic management of computing clusters and the experimental environment used in this work; in Section 4 we analyze the performance of the virtual cluster in the execution of high throughput computing (HTC) workloads; Section 5 reviews related work; finally, the paper ends with some conclusions in Section 6.
1. The OpenNebula Virtual Infrastructure Engine

Virtualization technology [12] decouples the virtual machine (a runtime environment, consisting of a guest OS and applications) from the physical resource. The main element in a virtual platform is the VM monitor (VMM) [13] or hypervisor (Xen, KVM, VMware, etc.), which allows multiple virtual systems running simultaneously on a single physical system (see figure 1).

![Figure 1. Virtualization technology.](image)

When the virtual infrastructure consists of a large amount of virtualized resources, the VM manager becomes a key component. A VM manager is responsible for the efficient management of the virtual infrastructure as a whole, by providing basic functionality for the deployment, control and monitoring of VMs on a distributed pool of physical resources. In this work we propose OpenNebula [4,5] as VM manager, as shown in figure 2.

![Figure 2. OpenNebula virtual manager engine.](image)

The OpenNebula virtual infrastructure engine provides the functionality needed to deploy, monitor and control VMs on a pool of distributed physical resources. Besides this basic functionality, OpenNebula allows:
• Balance of workload among physical resources to improve efficiency and utilization.
• Server consolidation to a reduced number of physical systems, so reducing space, administration effort, power and cooling requirements or supporting the shutdown of systems without interfering workload.
• Dynamic resizing of the physical infrastructure by adding new or shutting down existing hosts.
• Dynamic cluster partitioning to execute different services.
• Support for heterogeneous workloads with multiple (even conflicting) software requirements, allowing the execution of software with strict requirements as jobs that will only run with a specific version of a library or legacy application execution.
• On-demand provisioning of VMs, to adapt dynamically the infrastructure size to the service demands.

The architecture of OpenNebula has been designed to be flexible and modular to allow its integration with different hypervisors and infrastructure configurations. OpenNebula is composed of three main components: (i) The OpenNebula Core is a centralized component that manages the life-cycle of a VM by performing basic VM operations (e.g. deployment, monitoring, migration or termination). The core also provides a basic management and monitoring interface for the physical hosts. (ii) The Capacity Manager governs the functionality provided by the OpenNebula core. The capacity manager adjust the placement of VMs based on a set of pre-defined policies. The default capacity scheduler determines the best host to start a VM according to requirement and rank expressions consisting on infrastructure parameters. It also considers user-driven consolidation constraints. (iii) Virtualizer Access Drivers. In order to provide an abstraction of the underlying virtualization layer, OpenNebula uses pluggable drivers that exposes the basic functionality of the hypervisor (e.g. deploy, monitor or shutdown a VM). In this way, OpenNebula is not tied to any specific environment, so providing a uniform management layer regardless of the virtualization technology used.

OpenNebula can be easily integrated with cloud services, such as Amazon EC2, by using specific drivers. The Amazon EC2 driver translates a general VM deployment file in an EC2 instance description. The driver assumes that a suitable Amazon machine image (AMI) has been previously packed and registered in the S3 storage service, so when a given VM is to be deployed in EC2 its AMI counterpart is instantiated. The EC2 driver then converts the general requests made by OpenNebula core, such as deploy or shutdown, using the EC2 API. Figure 3 depicts the OpenNebula components and its interaction with the Amazon cloud.

2. Virtualization of Grid Infrastructures

Grid maintenance, operation and use exhibit many difficulties because of different reasons:

• High degree of hardware and software heterogeneity in the grid nodes. Such heterogeneity means an increase of the cost and length of the application development or porting cycle. New applications have to be tested in a great variety of environments where the developers have limited configuration capabilities. Application porting is currently one of the main obstacles for Grid adoption.
- Heterogeneity of environment requirements. Users often require specific versions of different software components (e.g. operating system, libraries or post-processing utilities). The cost of the installation, configuration and maintenance of user-specific or VO-specific worker nodes limits the flexibility of the infrastructure.

- Performance partitioning and isolation. Most of the computing infrastructures do not allow administrators to isolate and partition the performance of the physical resources they devote to different computing clusters or Grid infrastructures. This limits the quality of service and reliability of actual computing platforms, preventing a wide adoption of the Grid paradigm.

- High operational costs of deploying a grid infrastructure. The budget assigned to operations and maintenance activities in existing Grid infrastructures demonstrates the high cost of operating a Grid site, testing and deployment of new middleware distributions...

In order to overcome those barriers for a wider Grid adoption, there is an increasing interest on deploying grids on top of virtualized resources and cloud infrastructures [8]. The OpenNebula-based virtualization platform, along with cloud providers, can be used to virtualize, manage, and provision on-demand grid infrastructure components within a VO, such as individual hosts or computing clusters.

A computing cluster can be easily virtualized putting the front-end and worker nodes into VMs (see figure 4). The separation of resource provisioning, managed by OpenNebula, from job execution management, managed by existing batch queuing systems, provides the following benefits: (i) Elastic cluster capacity. The capacity of the cluster can be modified by deploying (or shutting down) virtual worker nodes on an on-demand basis, either in local physical resources or in remote EC2 resources. (ii) Cluster partitioning. The physical resources of the data center could be used to execute worker nodes bound to different virtual computing clusters, and thus isolating their workloads and partitioning the performance assigned to each virtual cluster. (iii) Heterogeneous configurations. The virtual worker nodes of a virtual cluster can have multiple (even conflicting) software configurations with a minimal operational cost, following an install once deploy
many approach. The above management is absolutely transparent to the user or the jobs that are being executed in the virtual cluster, as they are unaware of the physical resource (and its location) that is hosting each VM. So, the users and applications preserve their uniform view of all the virtual cluster nodes.

Grid middleware can operate in a transparent way on top of these virtualized computing resources, as shown in figure 5, bringing about the development of virtual grid infrastructures, which exhibits numerous benefits: easy support for VO-specific worker nodes, reduction of gridification cycles, dynamic balance of resources between VO’s, fault tolerance of key infrastructure components, easier deployment and testing of new middleware distributions, distribution of pre-configured components, cheaper development nodes, simplified training machines deployment, performance partitioning between local and grid services, etc.

The particular grid architecture shown in figure 5 is based on Globus Toolkit (GT) [9] middleware, including the GT basic services – Monitoring & Discovery System (MDS), Grid Resource Allocation and Management (GRAM) service, and GridFTP –, and the GridWay Metascheduler [10]. This middleware runs on top of a virtual computing infrastructure – a virtualized Sun Grid Engine (SGE) cluster, in this example –, which is managed by the OpenNebula virtual infrastructure engine. This virtualization layer is completely transparent to the services, applications or programming interfaces (such as DRMAA [11]) running on the grid.

3. An Architecture for the Elastic Management of Computing Clusters

In this section we illustrate how OpenNebula can be use to manage a computing cluster. OpenNebula provides a friendly user interface that allows to manage the virtual cluster in a simple, transparent way, and on an on-demand basis, adapting dynamically the cluster size to the variable computational demands throughout time. When the computational...
demand decreases, OpenNebula can be used to shutdown and consolidate virtual cluster nodes. Similarly, when the demand increases, OpenNebula can grow the cluster by deploying new virtual nodes on the local physical resource pool or hiring external virtual resources from Amazon EC2 in order to satisfy stronger computational demands or peak demand periods (see figure 6). This management is absolutely transparent for the user, who is unaware of what physical resource is hosting each virtual machine, or if this resource is local or remote, since the user has a uniform view of all the virtual nodes.

Figure 5. Virtual grid infrastructure.

Figure 6. Virtualization of a cluster with local and remote (Amazon EC2) computing nodes
The physical and virtual infrastructure used in this work for deploying the virtual computing cluster is shown in figure 7. The pool of physical hosts consists of five hosts (Host0 to Host4), which are interconnected by a private Gigabit Ethernet network (1000 Mbps). Each physical host node has a dual 2.0 GHz Xeon processor and 8GB of RAM. The Host0 acts as front-end of the physical pool and it is also connected to the Internet. This host runs the OpenNebula engine, which has the capacity of deploying, managing and monitoring local VMs on any host from the physical pool (using the XEN hypervisor) and also remote VMs on Amazon EC2.

The deployment of VMs (either local or remote) by OpenNebula can be controlled manually or can be done automatically by the scheduler module. In this case, the scheduling policy limits the number of VMs per physical host to a given threshold. When this limit is reached and if the cluster needs to grow, OpenNebula will deploy on-demand remote VMs, hosting a worker node, on Amazon EC2.

The virtual cluster consists of a front-end node and a variable set of workers nodes. Job submission and execution within the cluster is managed by SGE software. The virtual cluster front-end (SGE master host) has been deployed locally in the Host0 (since it needs to have Internet connectivity to be able to communicate with Amazon EC2 virtual machines). This cluster front-end acts also as NFS and NIS server for every worker node in the cluster.

The local virtual nodes (both front-end and workers), which are deployed with the XEN hypervisor, have a 32-bit i386 architecture (equivalent to 1.0 GHz Xeon processor), 512 MB of memory, and Debian Etch OS. The remote worker nodes, deployed on Amazon EC2, are based on an EC2 small standard instance (equivalent to 1.0-1.2 GHz Xeon processor), with 32-bit platform, 1.7 GB of memory, and Debian Etch OS.

Figure 8 shows the network infrastructure used for the virtual cluster. Every virtual worker node communicates with the front-end through the private local area network. The
local worker nodes and the front-end are connected to this private network by means of a virtual bridge configured in every physical host. On the other hand, the remote worker nodes (deployed on Amazon EC2) are connected to the private network by means of an OpenVPN tunnel, which is established between each remote node (OpenVPN clients) and the cluster front-end (OpenVPN server). With this configuration, every worker node (either local or remote) can communicate with the front-end and can use the common network services in a transparent way.

Figure 8. Network infrastructure for the virtual cluster.

4. Performance Evaluation

In this section, we present some application level benchmarks to study the behavior of the virtual cluster from the application point of view (see [14] for a detailed benchmarking of the Amazon Web Services). In particular, we will use the Embarrassingly Distributed (ED) benchmark from the NAS Grid Benchmarks [6,7] (NGB) suite. The ED benchmark models a typical high throughput computing application, which consists of multiple independent runs of the same program, but with different input parameters.

Let us first analyze the performance degradation introduced by the virtualization layer. Table 1 shows the results of running one iteration of the ED benchmark, for different problem sizes (classes A, B, and C), on a physical host and on a virtual machine deployed in the same physical host. As we can observe, the overhead of execution time due to virtualization is, in the worst case, around 15%.

In addition to virtualization overhead, communication latency can also cause significant performance degradation, specially when the front-end communicates with re-
Table 1. Execution times for the ED benchmark on physical and virtualized hosts

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Physical Host</th>
<th>Virtual Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>ED Class A</td>
<td>135</td>
<td>144</td>
</tr>
<tr>
<td>ED Class B</td>
<td>585</td>
<td>637</td>
</tr>
<tr>
<td>ED Class C</td>
<td>2432</td>
<td>2806</td>
</tr>
</tbody>
</table>

Remote worker nodes on the Amazon EC2, due to the intrinsic Internet delay, and the extra overhead introduced by the OpenVPN tunnel.

In order to quantify these communication latencies, figure 9 compares the response times experienced by local nodes and remote EC2 nodes when communicating with the cluster front-end using different network applications (ping, NFS service, and NIS service). It is clear that the server response time for remote nodes is, in all cases, significantly higher than the server response time for local nodes.

![Figure 9. Response times of different network applications.](image)

Similarly, figure 10 compares the access time for local and remote clients to various files of different size located in the NFS server. We can see again network latencies having strong impact on file transfer times.

In data-intensive applications, these delays (specially the NFS server response time) could degrade significantly the performance. However, computing clusters are mainly devoted for compute-intensive applications. In this kind of applications, these delays can affect mainly the initial and final stages of job execution, when access to the NIS account and NFS server is needed. To quantify how network latencies can affect the cluster performance, figure 11 shows the overall completion time of the entire benchmark (ED family, Class A, with 32 jobs) using a 8-node cluster, with different combinations of local and remote nodes. Considering that all the working nodes, either local or remote, have similar computing power, we can observe that the completion time for those configurations including remote nodes is around 25% higher than the configuration that uses only local nodes.
Considering cluster elasticity, it is also an important to prove that, in spite of the observed overheads, we can obtain an acceptable increment in the cluster performance when adding a growing number of remote nodes from the cloud provider to the cluster. Figure 12 shows the completion time of the entire benchmark (ED family, Class A, with 32 jobs) for a cluster with 4 local nodes and a growing number of remote nodes (from 0 to 8), and figure 13 shows the throughput (in jobs per minute) for the same cluster configurations. As we can observe, completion time can be reduced up to 43\% when adding 8 remote nodes to the cluster. Similarly, throughput achieved with 4L+8R configuration is 2.3 times higher than 4L+0R configuration.

5. Related Work

VM managers provide a centralized platform for the efficient management of virtual infrastructures, by automating the provisioning of virtual machines, and totally decoupling the physical resource pool, from the virtual infrastructure, and from the user service level. Most of VM managers are commercial and proprietary solutions, like Platform VM Orchestrator [15], VMware Virtual Center [16], Microsoft System Center Virtual Machine...
Manager [17], etc. Although there are also some open source initiatives like Enomalism [18], Ovirt [19], etc. These VM managers provide a high-level user interface that makes transparent to the user the underlying virtualization layer (XEN, KVM, VMware, etc.), and can also supply different tools and interfaces for creating and handling VM images, managing and monitoring physical resource pools, automatic provisioning of VMs, and VM re-allocating for consolidation and high availability support.

Compared to OpenNebula, the above-mentioned VM managers exhibit a monolithic and close structure, and can only operate, if any, with some preconfigured scheduling policies, which are, in general, very simple (usually based on CPU speed and CPU utilization). The open and flexible architecture provided by OpenNebula allows the definition of new heuristics for capacity provision. In this sense, Haizea [20] is an Open Source VM-based Lease Manager that has been integrated with OpenNebula [21] to offer resource leases, such as advance reservation leases, as a fundamental provisioning abstraction. In general, its modular architecture facilitates its integration with third-party components in the virtualization ecosystem, such as new virtualization platforms, cloud interfaces for remote management of virtual machines, service managers... Furthermore, most of those VM managers can only handle resources within the context of a single administrative domain, and can not interoperate with resources belonging to remote administrative domains (managed by a different VM manager) or belonging to a remote cloud
Finally, it is important to remark that OpenNebula provides network and contextualization support for the execution of complete services, consisting of groups of interconnected VMs. OpenNebula is one of the components being enhanced in the context of the Reservoir Project [22]. The aim of this project is to deliver complex IT services as utilities across different administrative domains, IT platforms and geographies.

Regarding on-demand provision of computational services, different approaches have been proposed in the literature. Traditionally, these methods consist in overlaying a custom software stack on top of an existing middleware layer, see for example the My-Cluster Project [23] or the Falkon system [24]. These approaches essentially shift the scalability issues from the application to the overlaid software layer, whereas the proposed solution transparently scales both the application and the computational cluster.

Also, the simultaneous use of heterogeneous configurations in the same cluster has been previously considered. These approaches usually integrate a local resource management system with VMs to provide on a per-job basis pre-configured execution environment, see for example [25]. A similar approach has been implemented at Grid level using the Globus GridWay Metascheduler [26,27].

The use of virtualization to provide on-demand clusters has been also studied in the context of the Globus Nimbus [28]. Nimbus provides a WSRF interface, functionally similar to that provided by Amazon EC2, to launch heterogeneous clusters on a remote. However, these clusters can not be easily integrated with the local resources nor can be supplemented with other cloud providers.

Finally, in a recent work BioTeam [29] has deployed the Univa UD UniCluster Express in an hybrid setup, that combines local physical nodes with virtual nodes deployed in the Amazon EC2. The work presented here integrates a VM manager in the local infrastructure as well, thus providing an elastic management not only for the outsourced resources but also for the local infrastructure.

6. Conclusions

In this work we have analyzed how Cloud computing can be used to support on-demand resource provisioning to provide elasticity in cluster and grid platforms. The integration of the cloud with the virtualization layer, managed by an efficient VM manager, allows us to give elastic capacity to the grid infrastructure using an external provider. This flexible approach, which separates the resource provisioning from the job execution management, provides important benefits: elastic cluster capacity to adapt the cluster to its dynamic workload; cluster partitioning to isolate it from other running services; and support for heterogeneous configurations tailored for each application class. To validate this architecture we have deployed a computing cluster on top of a virtualized infrastructure layer, which combines a local virtual machine manager (the OpenNebula engine) and a cloud resource provider (Amazon EC2). Although virtualization and communications introduce important overheads, which can have a negative impact on cluster performance, we have proved that performance degradation is very limited. Furthermore, from the point of view of the cluster elasticity, we have also proved that, in spite of the observed overheads, we can obtain a sustained performance increment when adding a growing number of remote nodes from the cloud provider to the cluster.
7. Acknowledgments

We would like to thank Javier Fontán, Tino Vázquez, and Luis González for their support to the development of the present work.

References

[22] The RESERVOIR Seed Team. RESERVOIR – An ICT Infrastructure for Reliable and Effective Delivery of Services as Utilities. IBM Research Report H-0262 (H0810-009), Haifa Research Laboratory, 2008
in conjunction with 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS-07), 2007


Clouds: An Opportunity for Scientific Applications?

Ewa DEELMAN\textsuperscript{a,1}, Bruce BERRIMAN\textsuperscript{b}, Gideon JUVE\textsuperscript{a}, Yang-Suk KEE\textsuperscript{c}, Miron LIVNY\textsuperscript{d}, Gurmeet SINGH\textsuperscript{a}

\textsuperscript{a} USC Information Sciences Institute, Marina del Rey, CA
\textsuperscript{b} Processing and Analysis Center & Michelson Science Center, California Institute of Technology, Pasadena, CA
\textsuperscript{c} Oracle US Inc
\textsuperscript{d} University of Wisconsin Madison, Madison, WI

Abstract. This paper examines issues related to the execution of scientific applications, and in particular computational workflows, on Cloud-based infrastructure. The paper describes the layering of application-level schedulers on top of the Cloud resources that enables grid-based applications to run on the Cloud. Finally, the paper examines issues of Cloud data management that supports workflow execution. We show how various ways of handling data have impact on the cost of the overall computations.

Keywords. Cloud computing, personal clusters, scientific workflows, data management in workflows, Amazon Cloud.

Introduction

Science applications today are becoming ever more complex. They are composed of a number of different application components, often written by different individuals and targeting a heterogeneous set of resources. The applications often involve many computational steps that may require custom execution environments. These applications also often process large amounts of data and generate large results. As the complexity of the scientific questions grows so does the complexity of the applications being developed to answer these questions.

Getting a result is only part of the scientific process. There are three other critical components of scientific endeavors: reproducibility, provenance, and knowledge sharing. We describe them in turn in the context of the scientific applications and revisit them towards the end of the chapter, evaluating how Clouds can meet these three challenges.

As the complexity of the applications increases, reproducibility [1, 2], the cornerstone of the scientific method, is becoming ever harder to achieve. Scientists often differentiate between scientific and engineering reproducibility. The former implies that another researcher can follow the same analytical steps, possibly on different data, and reach the same conclusions. Engineering reproducibility implies that

\textsuperscript{1} Corresponding author: deelman@isi.edu
one can reproduce the same result (on the same data with the same software) bit-by-bit. Reproducibility is hard to achieve because applications rely on a number of different software and different software versions (some at the system level and some at the application level) and access a number of data that can be distributed in the environment and can change over time (for example raw data may be calibrated in different ways as the understanding of the instrument behavior improves).

Reproducibility is only one of the critical components of the scientific method. As the complexity of the analysis grows, it becomes very difficult to determine how the data were created. This is especially complex when the analysis consists of a large-scale computation with thousands of tasks accessing hundred of data files. Thus the “capture and generation of provenance information is a critical part of the [...] generated data” [1].

Sharing of knowledge, of how to obtain particular results, of how to approach a particular problem, of how to calibrate the raw data, etc. are fundamental elements of educating new generations of scientists and of accelerating knowledge dissemination. When a new student joins a lab, it is important to quickly bring them up to speed, to teach him or her how to run a complex analysis on data being collected. When sharing results with a colleague, it is important to be able to describe exactly the steps that took place, which parameters were chosen, which software was used, etc.

Today sharing is difficult because of the complexity of the software and of how it needs to be used, of what parameters need to set, of what are the acceptable data to use, and of the complexity of the execution environment and its configuration (what systems support given codes, what message passing libraries to use, etc.).

Besides these over-reaching goals, applications also face computational challenges. Applications need to be able to take advantage of smaller, fully encapsulated components. They need to execute the computations reliably and efficiently while taking advantage of any number and type of resources including a local cluster, a shared cyberinfrastructure [3, 4], or the Cloud [5]. In all these environments there is a tradeoff between cost, availability, reliability, and ease of use and access.

One possible solution to the management of applications in heterogeneous execution environments is to structure the application as a workflow [6, 7] and let the workflow management system manage the execution of the application in different environments. Workflows enable the stitching of different computational tasks together and formalize the order in which the tasks need to execute. In astronomy, scientists are using workflows to generate science-grade mosaics of the sky [8], to examine the structure of galaxies [9] and in general to understand the structure of the universe. In bioinformatics, they are using workflows to understand the underpinnings of complex diseases [10, 11]. In earthquake science, workflows are used to predict the magnitude of earthquakes within a geographic area over a period of time [12]. In physics workflows are used to try to measure gravitational waves [13].

In our work, we have developed the Pegasus Workflow Management System (Pegasus-WMS) [14, 15] to map and executed complex scientific workflows on a number of different resources. In this context, the application is described in terms of logical components and logical data (independent of the actual execution environment) and the dependencies between the components. Since the application description is independent of the execution environment, mappings can be developed that can pick the right type of resources in a number of different execution environments [15], that can optimize workflow execution [16], and that can recover from execution failures [17, 18].
In this chapter we examine the issues of running workflow-based applications on the Cloud focusing on the costs incurred by an application when using the Cloud for computing and/or data storage. With the use of simulations, we evaluate the cost of running an astronomy application Montage [19] on the Cloud such as Amazon EC2/S3 [20].

1. The opportunity of the Cloud

Clouds have recently appeared as an option for on-demand computing. Originating in the business sector, Clouds can provide computational and storage capacity when needed, which can result in infrastructure savings for a business. For example, when a business invests in a given amount of computational capacity, buying servers, etc., they often need to plan for enough capacity to meet peak demands. This leaves the resources underutilized most of the time. The idea behind the Cloud is that businesses can plan only for a sustained level of capacity while reaching out to the Cloud resources in times of peak demand. When using the Cloud, applications pay only for what they use in terms of computational resources, storage, and data transfer in and out of the Cloud. In the extreme, a business can outsource all of its computing to the Cloud. Clouds are generally delivered by data centers strategically located in various energy-rich locations in the US and abroad. Because of the advances in network technologies, accessing data and computing across the wide area network is efficient from the point of view of performance. At the same time locating large-computing capabilities close to energy sources such as rivers, etc is efficient from the point of energy usage.

Today Clouds are also emerging in the academic arena, providing a limited number of computational platforms on demand: Nimbus [21], Eucalyptus [22], Cumulus [23], etc. These Science Clouds provide a great opportunity for researchers to test out their ideas and harden codes before investing more significant resources and money into the potentially larger-scale commercial infrastructure. In order to support the needs of a large number of different users with different demands on the software environment, Clouds are primarily built using resource virtualization technologies [24-27] that enable the hosting of a number of different operating systems and associated software and configurations on a single hardware host.

Clouds that provide computational capacities (Amazon EC2 [20], Nimbus [21], Cumulus [23], etc) are often referred as an Infrastructure as a Service (IaaS) because they provide the basic computing capabilities needed to deploy service. Other forms of Clouds include Platform as a Service (PaaS) that provide an entire application development environment and deployment container such as Google App Engine [28]. Finally, Clouds also provide complete services such as photo sharing, instant messaging [29], and many others, termed as Software as a Service (SaaS).

As already mentioned, commercial Clouds were built with business users in mind, however, scientific applications often have different requirements than enterprise customers. In particular, scientific codes often have parallel components and use MPI [30] or shared memory to manage the message-based communication between processors. More coarse-grained parallel applications often rely on a shared file system to pass data between processes. Additionally, as mentioned before, scientific applications are often composed of many inter-dependent tasks and consume and produce large amounts of data (often in the TeraByte range [12, 13, 31]). Today, these applications are running on the national and international cyberinfrastructure such as
the Open Science Grid [4], the TeraGrid [3], EGEE [32], and others. However, scientists are interested in exploring the capabilities of the Cloud for their work.

Clouds can provide benefits to today’s science applications. They are similar to the Grid, as they can be configured (with additional work and tools) to look like a remote cluster, presenting interfaces for remote job submission and data stage-in. As such scientists can use their existing grid software and tools to get their work done. Another interesting aspect of the Cloud is that by default it includes resource provisioning as part of the usage mode. Unlike the Grid, where jobs are often executed on a best-effort basis, when running on the Cloud, a user requests a certain amount of resources and has them dedicated for a given duration of time. (An open question in today’s Clouds is how many resources and how fast can anyone request at any given time.) Resource provisioning is particularly useful for workflow-based applications, where overheads of scheduling individual, inter-dependent tasks in isolation (as it is done by Grid clusters) can be very costly. For example, if there are two dependent jobs in the workflow, the second job will not be released to a local resource manager on the cluster until the first job successfully completes. Thus the second job will incur additional queuing time delays. In the provisioned case, as soon as the first job finishes, the second job is released to the local resource manager and since the resource is dedicated, it can be scheduled right away. Thus the overall workflow can be executed much more efficiently.

Virtualization also opens up a greater number of resources to legacy applications. These applications are often very brittle and require a very specific software environment to execute successfully. Today, scientists struggle to make the codes that they rely on for weather prediction, ocean modeling, and many other computations to work on different execution sites. No one wants to touch the codes that have been designed and validated many years ago in fear of breaking their scientific quality. Clouds and their use of virtualization technologies may make these legacy codes much easier to run. Now, the environment can be customized with a given OS, libraries, software packages, etc. The needed directory structure can be created to anchor the application in its preferred location without interfering with other users of the system. The downside is obviously that the environment needs to be created and this may require more knowledge and effort on the part of the scientist then they are willing or able to spend.

In this chapter, we focus on a particular Cloud, Amazon EC2 [20]. On Amazon, a user requests a certain number of a certain class of machines to host the computations. One also can request storage on the Amazon S3 storage system. This is a fairly basic environment in which virtual images need to deployed and configured. Virtual images are critical to making Clouds such as Amazon EC2 work. One needs to build an image with the right operating system, software packages etc. and then store them in S3 for deployment. The images can also contain the basic grid tools such as Condor [33], Globus [34], higher-level software tools such as workflow management systems (for example Pegasus-WMS [14]), application codes, and even application data (although this is not always practical for data-intensive science applications). Science applications often deal with large amounts of data. Although EC2-like Clouds provide 100-300GB of local storage that is often not enough, especially since it also needs to hosts the OS and all other software. Amazon S3 can provide additional long-term storage with simple put/get/delete operations. The drawback to S3 for current grid applications is that it does not provide any grid-like data access such as GridFTP [35]. Once an image is built it can be easily deployed at any number of locations. Since the
environment is dynamic and network IPs are not known beforehand, dynamic configuration of the environment is key. In the next section we describe a technology that can manage multiple virtual machines and configure them as a Personal Cluster.

2. Managing applications on the Cloud

In recent years, a number of technologies have emerged to manage the execution of applications on the Cloud. Among them are Nimbus [21] with its virtual cluster capabilities and Eucalyptus with its EC2-like interfaces [22]. Here, we describe a system that allows a user to build a Personal Cluster that can bridge the Grid and Cloud domains and provide a single point of entry for user jobs.

2.1. Personal cluster

Best-effort batch queuing has been the most popular resource management paradigm used for high-performance scientific computing. Most clusters in production today employ a variety of batch systems such as SGE (Sun Grid Engine) [36], PBS (Portable Batch System) [37], Condor [38], LSF (Load Sharing Facility) [39], and so on for efficient resource management and QoS (Quality of Service). Their major goal is to manage complex workloads on complex systems, achieving high-throughput, and maximizing system utilization. In the meantime, we are facing a new computing paradigm based on virtualization technologies such as virtual cluster and compute Clouds for parallel and distributed computing. This new paradigm provisions resources on demand and enables easy and efficient resource management for application developers. However, scientists commonly have difficulty in developing and running their applications, fully exploiting the potential of a variety of paradigms because the new technologies introduce additional complexity to the application developers and users. In this sense, configuring a common execution environment automatically on behalf of users regardless of local computing environments can lessen the burden of application development significantly. The Personal Cluster was proposed to pursue this goal.

The Personal Cluster [40] is a collection of computing resources controlled by a private resource manager, instantiated on demand from a resource pool in a single administrative domain such as batch resources and compute clouds. The Personal Cluster deploys a user-level resource manager to a partition of resources at runtime. The Personal Cluster resides on the resources for a specified time period on the behalf of the user and provides a uniform computing environment, taking the place of local resource managers. As a result, the Personal Cluster gives an illusion to the user that the instant cluster is dedicated to the user during the application’s lifetime and that she/he has a homogeneous computing environment regardless of local resource management paradigms. Figure 1 illustrates the concept of Personal Cluster. Regardless of whether resources are managed by a batch scheduler or a Cloud infrastructure, the Personal Cluster instantiates a private cluster only for the user, configured with a dedicated batch queue (i.e. PBS) and a web services (i.e., WS-GRAM [41]) on-the-fly. Once a Personal Cluster instance is up and running, the user can run his/her application by submitting jobs into the private queue directly.
Scientists can benefit from the Personal Cluster in a variety of aspects. First, the Personal Cluster provides a uniform job/resource management environment over heterogeneous resources regardless of system-level resource management paradigms. For instance, to execute a job on batch resources, the users have to write a job submission script. If users want to run their applications on heterogeneous resources such as TeraGrid [42], they have to write multiple job descriptions for each resource. Similarly, users need to run individual jobs on each processor using the secure shell tools such as ssh and scp for compute Clouds. The Personal Cluster lessens this burden for the user by providing a uniform runtime environment regardless of local resource management software. That is, the commodity batch scheduler installed for the allocated resources makes the execution environment homogeneous and consistent.

Second, the Personal Cluster can provide QoS of resources when using space-sharing batch resources. The common interest of scientists is to achieve the best performance of their applications in a cost-effective way. However, space-sharing batch systems are unlikely to optimize the turnaround time of a single application especially those consisting of multiple tasks against the fair sharing of resources between jobs. For the best-effort resource management, tasks submitted for an application have to compete for resources with other applications. In consequence, the execution time of an application that consists of multiple jobs (e.g., workflows, parameter studies) is unpredictable because other applications can interrupt the jobs and thus the progress of application. If an application is interrupted by a long-running job, the overall turnaround time of the application can be delayed significantly. In order to prevent the performance degradation due to such interruptions, the user can cluster the tasks together and submit a single script that runs the actual tasks when the script is executed. However, this clustering technique cannot be benefited by the common capabilities for efficient scheduling such as backfilling provided by resource management systems. By contrast, the Personal Cluster can have an exclusive access to the resource partition during the application’s lifetime once local resource managers allocate resource partitions. In addition, the private batch scheduler of Personal Cluster can optimize the execution of application tasks.

Third, the Personal Cluster enables a cost-effective resource allocation. Since the Personal Cluster acquires resources via default local resource allocation strategy and releases them immediately at termination, it requires neither modifications of local
schedulers nor extra cost for reservation. In the sense that a resource partition is
dedicated for the application, a user-level advance reservation is a promising solution
to secure performance [43]. However, the user-level advance reservation is still neither
popular nor cheap in general because it adversely affects the fairness and the efficient
resource utilization. In addition, user-level advance reservation can be cost-ineffective
because users have to pay for the entire reservation period regardless of whether they
use the resources or not. Resource providers may charge users more for reservations
since they can have an adverse effect on the resource utilization of the entire system
and the fairness between jobs. By contrast, the Personal Cluster can have the same
benefits without the resources having any special scheduler like advance reservation.
The Personal Cluster does not cause any surcharge for reservations since the resources
are allocated in a best-effort manner. Moreover, they can terminate at any time without
any penalty because the allocated resources will be returned immediately at termination.

Finally, the Personal Cluster leverages commodity tools. A resource manager is
not only a placeholder for the allocated resources but also a gateway taking care of
resource-specific accesses as well as task launching and scheduling. It is redundant and
unnecessary to implement a new job/resource manager for this purpose. As an
alternative, the Personal Cluster employs commodity tools. They provide a vehicle for
efficient resource management and make the application development simple.

The current implementation of Personal Cluster is based on the WS-based Globus
Toolkit [44] and a PBS [37] installation. The Personal Cluster uses the similar
mechanism to Condor glidein [45]. Once a system-level resource manager allocates a
partition of resources, a user-level PBS scheduled on the resources holds the resources
for a user-specified time and a user-level WS-GRAM (configured at runtime for the
partition) accepts jobs from the user and relays them to the user-level PBS. As a result,
users can bypass the system-level resource manager and benefit from the low
scheduling overhead with the private scheduler.

2.2. Personal Cluster on batch resources

A barrier to instantiating a Personal Cluster on batch-controlled resources is the
network configuration of the cluster such as firewall, access control, etc. The Personal
Cluster assumes a relatively conservative configuration where a remote user can access
the clusters via public gateway machines while the individual nodes behind batch
systems are private and the accesses to the allocated resources are allowed only during
the time period of resource allocation. Then, a batch scheduler allocates a partition of
resources and launches the placeholders for Personal Cluster on the allocated resources
via remote launching tools such as rsh, ssh, pbsdsh, mpiexec, etc, depending on the
local administrative preference. Thus, the security of the Personal Cluster relies on
what is provided by local systems.

A client component called PC factory instantiates Personal Clusters on the behalf of
user, submitting resource requests to remote batch schedulers, monitoring the status of
resource allocation process, and setting up default software components. In essence, the
actual job the factory submits sets up a private, temporary version of PBS on a per
application basis. This user-level PBS installation has access to the resources and
accepts the application jobs from the user. As a foundation software, the Personal
Cluster uses the most recent open source Torque package [46] and has made several
source level modifications to enable a user-level execution. In theory, this user-level
PBS can be replaced with other resource managers running at the user-level.
Figure 2 illustrates how to configure a personal cluster using the user-level PBS and WS-GRAM service when the resources are under the control of a batch system and Globus Toolkits based on Web Services (i.e., GT4) provide the access mechanisms. A user-level GRAM server and a user-level PBS are preinstalled on the remote cluster and the user-level GRAM-PBS adaptor is configured to communicate with the user-level PBS. The PC factory first launches a kick-start script to identify the allocated resources and then invokes a bootstrap script for configuring the PBS daemons on each node. The kick-start script assigns an ID for each node, not each processor, and identifies the number of processors allocated for each node. For batch resources, a system-level batch scheduler will launch this kick-start script on the resources via a system-level GRAM adaptor (e.g., GRAM-PBS, GRAM-LSF). If a local resource manager does not have any mechanism to launch the kick-start script on the individual resources, the PC factory launches it one by one using ssh. Once the kick-start script has started successfully, the system-level batch scheduler retreats and the PC factory regains the control of the allocated resources. At last, the bootstrap script configures the user-level PBS for the resources on a per-node basis. The node with ID 0 hosts a PBS server (i.e., pbs_server) and a PBS scheduler (i.e., pbs_sched) while the others host PBS workers (i.e., pbs_mom). In the meantime, the bootstrap script creates the default directories for log, configuration files, and so on; generates a file for the communication with the personal GRAM-PBS adaptor (i.e., globus-pbs.conf), configures the queue management options; and starts the daemon executables, based on its role. Finally, the PC factory starts a user-level WS-GRAM server via the system-level GRAM-FORK adaptor on a gateway node of the resources.

Once the user-level PBS and GRAM are in production, the user can bypass the system-level scheduler and utilize the resources as if a dedicated cluster was available. Now a personal cluster is ready and the user can submit application jobs via the private,
A personal cluster is instantiated on compute Clouds in a way similar to that of the batch resources. However, since the virtual processors from the Cloud are instantiated dynamically, the Personal Cluster needs to deal with issues due to the system information determined at runtime such as hostname and IP.

The PC factory first constructs a physical cluster with the default system and network configurations. The PC factory boots a set of virtual machines by picking a preconfigured image from the virtual machine image repository. When all virtual machines are successfully up and running, the factory weaves them with NFS (Network File System). Specifically, only the user working directory is shared among the participated virtual processors. Then, the factory registers all virtual processors as known host and shares the public key and private key of the user for secure shell so the user can access every virtual processor using the ssh without password. It also generates an MPI (Message Passing Interface) machine file for the participating processors. Finally, the factory disables remote access to all processors except one that plays as a gateway node. The user can access the Personal Cluster instance through the user-level PBS and WS-GRAM setup on the gateway node.

One critical issue is to have a host certificate for the WS-GRAM service. A node hosting the GRAM service needs a host certificate based on host name or IP for the user to be able to authenticate the host. However, the hostname and IP of a virtual processor is dynamically determined at runtime. As such, we cannot obtain a host certificate for a virtual processor permanently, which implies that the system-level GRAM service cannot be setup for clouds dynamically. Instead, we use the self authentication method so that the factory starts the WS-GRAM service using the user’s certificate without setting up host certificate. A user certificate can be imported into the virtual processors by using the myproxy service. The secure shell access with password and Globus self-authentication method enable only the user to access and use the Personal Cluster instance. Once this basic configuration is completed, the factory repeats the same process for batch resources and setups up the user-level PBS and WS-GRAM service.

3. Montage application

So far, we focused on the technology-side of the equation. In this section, we examine a single application, which is a very important and popular astronomy application. We use the application as a basis of evaluating the cost/performance tradeoffs of running applications on the Cloud. It also allows us to compare the cost of the Cloud for generating science products as compared to the cost of using your own compute infrastructure.
3.1. What Is Montage and Why Is It Useful?

Montage [8] is a toolkit for aggregating astronomical images into mosaics. Its scientific value derives from three features of its design [47]:

- It preserves the calibration and astrometric fidelity of the input images to deliver mosaics that meet user-specified parameters of projection, coordinates, and spatial scale. It supports all projections and coordinate systems in use in astronomy.
- It contains independent modules for analyzing the geometry of images on the sky, and for creating and managing mosaics; these modules are powerful tools in their own right and have applicability outside mosaic production, in areas such as data validation.
- It is written in American National Standards Institute (ANSI)-compliant C, and is portable and scalable – the same engine runs on desktop, cluster, supercomputer or cloud environments running common Unix-based operating systems such as Linux, Solaris, Mac OS X and AIX.

The code is available for download for non-commercial use [48]. The current distribution, version 3.0, includes the image mosaic processing modules and executives for running them, utilities for managing and manipulating images, and all third-party libraries, including standard astronomy libraries for reading images. The distribution also includes modules for installation of Montage on computational grids. A web-based Help Desk is available to support users, and documentation is available on-line, including the specification of the Applications Programming Interface (API).

Montage is highly scalable. It uses the same set of modules to support two instances of parallelization: MPI [49], a library specification for message passing, and Planning and Execution for Grids (Pegasus), a toolkit that maps workflows on to distributed processing environments [18]. Parallelization and performance are described in detail in [50, 51].

Montage is in active use in generating science data products, in underpinning quality assurance and validation of data, in analyzing scientific data and in creating Education and Public Outreach products [52].

3.2. Montage Architecture and Algorithms

3.2.1. Supported File Formats

Montage supports two-dimensional images that adhere to the definition of the Flexible Image Transport System (FITS) standard [53], the international standard file format in astronomy. The relationship between the pixel coordinates in the image and physical units is defined by the World Coordinate System (WCS) [53]. Included in the WCS is a definition of how celestial coordinates and projections are represented in the FITS format as keyword=value pairs in the file headers. Montage analyzes these pairs of values to discover the footprints of the images on the sky and calculates the footprint of the image mosaic that encloses the input footprints. Montage supports all projections supported by WCS, and all common astronomical coordinate systems. The output mosaic is FITS-compliant, with the specification of the image parameters written as keywords in the FITS header.
3.2.2. Design Philosophy

There are four steps in the production of an image mosaic. They are illustrated as a flow chart in Figure 3, which shows where the processing can be performed in parallel:

- Discover the geometry of the input images on the sky, labeled “image” in Figure 3, from the input FITS keywords and use it to calculate the geometry of the output mosaic on the sky.
- Re-project the flux in the input images to conform to the geometry of the output geometry of the mosaic, as required by the user-specified spatial scale, coordinate system, WCS-projection, and image rotation.
- Model the background radiation in the input images to achieve common flux scales and background level across the mosaic. This step is necessary because there is no physical model that can predict the behavior of the background radiation. Modeling involves analyzing the differences in flux levels in the overlapping areas between images, fitting planes to the differences, computing a background model that returns a set of background corrections that forces all the images to a common background level, and finally applying these corrections to the individual images. These steps are labeled “Diff,” “Fitplane,” “BgModel,” and “Background” in Figure 3.
- Co-add the re-projected, background-corrected images into a mosaic.

Each production step has been coded as an independent engine run from an executive script. This toolkit design offers flexibility to users. They may, for example, use Montage as a re-projection tool, or deploy a custom background rectification algorithm while taking advantage of the re-projection and co-addition engines.

3.3. An On-Demand Image Mosaic Service

The NASA/IPAC Infrared Science Archive [54] has deployed an on-request image mosaic service. It uses low cost, commodity hardware with portable, Open Source software, and yet is fault-tolerant, scalable, extensible and distributable. Users request a mosaic on a simple web form [55]. The service returns mosaics from three wide-area survey data sets: the 2-Micron All-Sky Survey (2MASS), housed at the NASA IPAC Infrared Science Archive (IRSA), the Sloan Digital Sky Survey (SDSS), housed at FermiLab, and the Digital Sky Survey (DSS), housed at the Space Telescope Science Institute (STScI). The first release of the service restricts the size of the mosaics to 1 degree on a side in the native projections of the three datasets. Users may submit any number of jobs, but only ten may run simultaneously and the mosaics will be kept for only 72 hours after creation. These restrictions will be eased once the operational load on the service is better understood. The return page shows a JPEG of the mosaic, and provides download links for the mosaic and an associated weighting file. Users may monitor the status of all their jobs on a web page that is refreshed every 15 seconds, and may request e-mail notification of the completion of their jobs. A possible solution for supporting a larger number of mosaic requests is to leverage resources provided by today’s clouds.
4. Issues of running workflow applications on the Cloud

Today applications such as Montage are asking: What are Clouds? How do I run on them? How do I make good use of my funds? Often, domain scientists have heard about Clouds but have no good idea of what they are, how to use them, and how much would Cloud resources cost in the context of an application. In this section we posed three cost-related questions (a more detailed study is presented in [56]):

1. How many resources do I allocate for my computation or my service?
2. How do I manage data within a workflow in my Cloud applications?
3. How do I manage data storage—where do I store the input and output data?

We picked the Amazon services [57] as the basic model. Amazon provides both compute and storage resources on a pay-per-use basis. In addition it also charges for transferring data into the storage resources and out of it. As of the writing of this chapter, the basic charging rates were:

- $0.15 per GB-Month for storage resources
- $0.1 per GB for transferring data into its storage system
- $0.16 per GB for transferring data out of its storage system
- $0.1 per CPU-hour for the use of its compute resources.

There is no charge for accessing data stored on the Amazon storage systems by tasks running on its compute resources. Even though as shown above, some of the quantities span over hours and months, in our experiments we normalized the costs on a per second basis. Obviously, service providers charge based on hourly or monthly usage, but here we assume cost per second. The cost per second corresponds to the case where there are many analyses conducted over time and thus resources are fully utilized.

In this chapter, we use the following terms: application—the entity that provides a service to the community (the Montage project), user request—a mosaic requested by the user from the application, the Cloud—the computing/storage resource used by the application to deliver the mosaic requested by the user.
Figure 4 illustrates the concept of cloud computing as could be implemented for the use by an application. The user submits a request to the application, in the case of Montage via a portal. Based on the request, the application generates a workflow that has to be executed using either local or cloud computing resources. The request manager may decide which resources to use. A workflow management system, such as Pegasus [15], orchestrates the transfer of input data from image archives to the cloud storage resources using appropriate transfer mechanisms (the Amazon S3 storage resource supports the REST and HTTP transfer protocol [58]). Then, compute resources are acquired and the workflow tasks are executed over them. These tasks can use the cloud storage for storing temporary files. At the end of the workflow, the workflow system transfers the final output from the cloud storage resource to a user-accessible location.

In order to answer the questions raised at the beginning of this section, we performed simulations. No actual provisioning of resources from the Amazon system was done. Simulations allowed us to evaluate the sensitivity of the execution cost to workflow characteristics such as the communication to computation ratio by artificially changing the data set sizes. This would have been difficult to do in a real setting. Additionally, simulations allow us to explore the performance/cost tradeoffs without paying for the actual Amazon resources or incurring the time costs of running the actual computation. The simulations were done using the GridSim toolkit [59]. Certain custom modifications were done to perform accounting of the storage used during the workflow execution.

We used three Montage workflows in our simulations:
1. Montage 1 Degree: A Montage workflow for creating a 1 degree square mosaic of the M17 region of the sky. The workflow consists of 203 application tasks.
2. Montage 4 Degree: A Montage workflow for creating a 4 degree square mosaic of the M17 region of the sky. The workflow consists of 3,027 application tasks.

These workflows can be created using the mDAG [60] component in the Montage distribution [61]. The workflows created are in XML format. We wrote a program for
parsing the workflow description and creating an adjacency list representation of the graph as an input to the simulator. The workflow description also includes the names of all the input and output files used and produced in the workflow. The sizes of these data files and the runtime of the tasks were taken from real runs of the workflow and provided as additional input to the simulator.

We simulated a single compute resource in the system with the number of processors greater than the maximum parallelism of the workflow being simulated. The compute resource had an associated storage system with infinite capacity. The bandwidth between the user and the storage resource was fixed at 10 Mbps. Initially all the input data for the workflow are co-located with the application. At the end of the workflow the resulting mosaic is staged out to the application/user and the simulation completes. The metrics of interest that we determine from the simulation are:

1. The workflow execution time.
2. The total amount of data transferred from the user to the storage resource.
3. The total amount of data transferred from the storage resource to the user.
4. The storage used at the resource in terms of GB-hours. This is done by creating a curve that shows the amount of storage used at the resource with the passage of time and then calculating the area under the curve.

We now answer the questions we posed in our study.

4.1. How many resources do I allocate for my computation or my service?

Here we examine how best to use the cloud for individual mosaic requests. We calculate how much would a particular computation cost on the cloud, given that the application provisions a certain number of processors and uses them for executing the tasks in the application. We explore the execution costs as a function of the number of resources requested for a given application. The processors are provisioned for as long as it takes for the workflow to complete. We vary the number of processors provisioned from 1 to 128 in a geometric progression. We compare the CPU cost, storage cost, transfer cost, and total cost as the number of processors is varied. In our simulations we do not include the cost of setting up a virtual machine on the cloud or tearing it down, this would be an additional constant cost.

The Montage 1 degree square workflow consists of 203 tasks and in this study the workflow is not optimized for performance. Figure 5 shows the execution costs for this workflow. The most dominant factor in the total cost is the CPU cost. The data transfer costs are independent of the number of processors provisioned. The figure shows that the storage costs are negligible as compared to the other costs. The Y-axis is drawn in logarithmic scale to make the storage costs discernable. As the number of processors is increased, the storage costs decline but the CPU costs increase. The storage cost declines because as we increase the number of processors, we need them for shorter duration since we can get more work done in parallel. Thus we also need storage for shorter duration and hence the storage cost declines. However, the increase in the CPU cost far outweighs any decrease in the storage costs and as a result the total costs also increase with the increase in the number of provisioned processors. The total costs shown in the graphs are aggregated costs for all the resources used.

Based on Figure 5, it would seem that provisioning the least amount of processors is the best choice, at least from the point of view of monetary costs (60 cents for the 1 processor computation versus almost 4$ with 128 processors). However, the drawback
in this case is the increased execution time of the workflow. Figure 5 (right) shows the execution time of the Montage 1 Degree square workflow with increasing number of processors. As the figure shows, when only one processor is provisioned leading to the least total cost, it also leads to the longest execution time of 5.5 hours. The runtime on 128 processors is only 18 minutes. Thus a user who is also concerned about the execution time, faces a trade-off between minimizing the execution cost and minimizing the execution time.

Figure 5: Cost of One Degree Square Montage on the Cloud.

Figure 6 shows similar results for the Montage 4 degree workflow as for the 1 degree Montage workflow. The Montage 4 degree square workflow consists of 3,027 application tasks in total. In this case running on 1 processor costs $9 with a runtime of 85 hours; with 128 processors, the runtime decreases to 1 hour with a cost of almost $14. Although the monetary costs do not seem high, if one would like to request many mosaics to be done, as would be in the case of providing a service to the community, these costs can be significant. For example, providing 500 4-degree square mosaics to astronomers would cost $4,500 using 1 processor versus $7,000 using 128 processors. However, the turnaround of 85 hours may be too much to take by a user. Luckily, one does not need to consider only the extreme cases. If the application provisions 16
processors for the requests, the turnaround time for each will be approximately 5.5 hours with a cost of $9.25, and thus a total cost of 500 mosaics would be $4,625, not much more than in the 1 processor case, while giving a relatively reasonable turnaround time.

4.2. How do I manage data within a workflow in my Cloud applications?

For this question, we examine three different ways of managing data within a workflow. We present three different implementation models that correspond to different execution plans for using the Cloud storage resources. In order to explain these computational models we use the example workflow shown in Figure 7. There are three tasks in the workflow numbered from 0 to 2. Each task takes one input file and produces one output file.

![Figure 7. An Example Workflow.](image)

We explore three different data management models:

1. **Remote I/O (on-demand):** For each task we stage the input data to the resource, execute the task, stage out the output data from the resource and then delete the input and output data from the resource. This is the model to be used when the computational resources used by the tasks have no shared storage. For example, the tasks are running on hosts in a cluster that has only a local file system and no network file system. This is also equivalent to the case where the tasks are doing remote I/O instead of accessing data locally. Figure 8 (a) shows how the workflow from Figure 7 looks like after the data management tasks for the Remote I/O are added by the workflow management system.

2. **Regular:** When the compute resources used by the tasks in the workflow have access to shared storage, it can be used to store the intermediate files produced in the workflow. For example, once task 0 (Figure 8b) has finished execution and produced the file $b$, we allow the file $b$ to remain on the storage system to be used as input later by tasks 1 and 2. In fact, the workflow manager does not delete any files used in the workflow until all the tasks in the workflow have finished execution. After that file $d$ which is the net output of the workflow is staged out to the application/user and all the files $a$–$c$ are deleted from the storage resource. As mentioned earlier this execution mode assumes that there is shared storage that can be accessed from the compute resources used by the tasks in the workflow. This is true in the case of the Amazon system where the
data stored in the S3 storage resources can be accessed from any of the EC2 compute resources.

3. **Dynamic cleanup**: In the regular mode, there might be files occupying storage resources even when they have outlived their usefulness. For example file $a$ is no longer required after the completion of task 0 but it is kept around until all the tasks in the workflow have finished execution and the output data is staged out. In the dynamic cleanup mode, we delete files from the storage resource when they are no longer required. This is done for example in Pegasus by performing an analysis of data use at the workflow level [62]. Thus file $a$ would be deleted after task 0 has completed, however file $b$ would be deleted only when task 2 has completed (Figure 8c). Thus the dynamic cleanup mode reduces the storage used during the workflow and thus saves money. Previously, we have quantified the improvement in the workflow data footprint when dynamic cleanup is used for data-intensive applications similar to Montage [63]. We found that dynamic cleanup can reduce the amount of storage needed by a workflow by almost 50%.

Here we examine the issue of the cost of user requests for scientific products when the application provisions a large number of resources from the Cloud and then allows the request to use as many resources as it needs. The application is in this scenario responsible for scheduling the user requests onto the provisioned resources (similarly to the Personal Cluster approach). In this case, since the processor time is used only as much as needed, we would expect that the data transfer and data storage costs may play a more significant role in the overall request cost. As a result, we examine the tradeoffs between using three different data management solutions: 1) remote I/O, where tasks access data as needed, 2) regular, where the data are brought in at the beginning of the
computation and they and all the results are kept for the duration of the workflow, and
3) cleanup, where data no longer needed are deleted as the workflow progresses. In the
following experiments we want to determine the relationship between the data transfer
cost and the data storage cost and compare it to the overall execution cost.

Figure 9 (left) shows the amount of storage used by the workflow in the three
modes in space-time units for the 1 degree square Montage Workflow. The least
storage used is in the remote I/O mode since the files are present on the resource only
during the execution of the current task. The most storage is used in the regular mode
since all the input data transferred and the output data generated during the execution of
the workflow is kept on the storage until the last task in the workflow finishes
execution. Cleanup reduces the amount of storage used in the regular mode by deleting
files when they are no longer required by later tasks in the workflow.

Figure 9 (middle) shows the amount of data transfer involved in the three
execution modes. Clearly the most data transfer happens in the remote I/O mode since
we transfer all input files and transfer all output files for each task in the workflow.
This means that if the same file is being used by more than one job in the workflow in
the remote I/O mode the file may be transferred in multiple times whereas in the case
of regular and cleanup modes, the file would be transferred only once. The amount of
data transfer in the Regular and the Cleanup modes are the same since dynamically
removing data at the execution site does not affect the data transfers. We have
categorized the data transfers into data transferred to the resource and data transferred
out of the resource since Amazon has different charging rates for each as mentioned
previously. As the figure shows, the amount of data transferred out of the resource is
the same in the Regular and Cleanup modes. The data transferred out is the data of
interest to the user (the final mosaic in case of Montage) and it is staged out to the user
location. In the Remote I/O mode intermediate data products that are needed for
subsequent computations but are not of interest to the user also need to be staged-out to
the user-location for future access. As a result, in that mode the amount of data being
transferred out is larger than in the other two execution strategies.

Figure 9 (right) shows the costs (in monetary units) associated with the execution
of the workflow in the three modes and the total cost in each mode. The storage costs
are negligible as compared to the data transfer costs and hence are not visible in the
figure. The Remote I/O mode has the highest total cost due to its higher data transfer
costs. Finally, the Cleanup mode has the least total cost among the three. It is important
to note that these results are based on the charging rates currently used by Amazon. If
the storage charges were higher and transfer costs were lower, it is possible that the
Remote I/O mode would have resulted in the least total cost of the three.

![Figure 9: Data Management Costs for the 1 degree square Montage.](image-url)
Figure 10 shows the metrics for the Montage 4 degrees square workflow. The cost distributions are similar to the smaller workflow and differs only in magnitude as can be seen from the figures.

![Figure 10: Data Management Costs for the 4 degree square Montage.](image)

We also wanted to quantify the effect of the different workflow execution modes on the overall workflow cost. Figure 11 shows these total costs. We can see that there is very little difference in cost between the Regular and Cleanup mode, thus if space is not an issue, cleaning up the data alongside the workflow execution is not necessary. We also notice that the cost of Remote I/O is much greater because of the additional cost of data transfer.

![Figure 11: Overall Workflow Cost for Different Data Management Strategies.](image)

4.3. How do I manage data storage—where do I store the input and output data?

In the study above we assumed that the main data archive resided outside of the Cloud and that when a mosaic was being computed, only that data was being transferred to the Cloud. We also wanted to ask the question whether it would make sense to store the data archive itself on the Cloud. The 2Mass archive that is used for the mosaics takes up approximately 12TB of storage which on Amazon would cost $1,800 per month. Calculating a 1 degree square mosaic and delivering it to the user costs $2.22 when the archive is outside of the Cloud. When the input data is available on S3, the cost of the mosaic goes down to $2.12. Therefore to overcome the storage costs, users would need to request at least $1,800/($2.22-$2.12) = 18,000 mosaics per month which is high for today’s needs. Additionally, the $1,800 cost does not include the initial cost of transferring data into the Cloud which would be an extra $1,200.
Is the $1,800 cost of storage reasonable as compared to the amount spent by the Montage project? If we add up the cost of storing the archive data on S3 over three years, it will cost approximately $65,000. This cost does not include access to the data from outside the Cloud. Currently, the Montage project is spending approximately $15,000 over three years for 12TB of storage. This includes some labor costs but does not include facility costs such as space, power, etc. Still it would seem that the cost of storage of data on the Cloud is quite expensive.

5. Conclusions

In this chapter we took a first look at issues related to running scientific applications on Cloud. In particular we focused on the cost of running the Montage application on the Amazon Cloud. We used simulations to evaluate these costs. We have seen that there exists a classic tradeoff between the runtime of the computation and its associated cost and that one needs to find a point at which the costs are manageable while delivering performance that can meet the users’ demands. We also demonstrated that storage on the Cloud can be costly. Although this cost is minimal when compared to the CPU cost of individual workflows, over time the storage cost can be significant.

Clouds are still in their infancy--there are only a few commercial and academic providers. As the field matures, we expect to see a more diverse selection of fees and quality of service guarantees for the different resources and services provided by Clouds. It is possible that some providers will have a cheaper rate for compute resources while others will have a cheaper rate for storage and provide a range of quality of service guarantees. As a result, applications will have more options to consider and more execution and provisioning plans to develop to address their computational needs.

Many other aspects of the problem still need to be addressed. These include the startup cost of the application on the cloud, which is composed of launching and configuring a virtual machine and its teardown, as well as the often one-time cost of building a virtual image suitable for deployment on the cloud. The complexity of such an image depends on the complexity of the application. We also did not explore other cloud issues such as security and data privacy. The reliability and availability of the storage and compute resources are also an important concern.

The question exists whether scientific applications will move into the Cloud. Clearly, there is interest in the new computational platform, the promise of on-demand, pay-as-you-go resources is very attractive. However, much needs to be done to make Clouds accessible to a scientist. User-friendly tools need to be developed to manage the Cloud resources and to configure them in a way suitable for a scientific application. Easy to use tools need to be developed to help build and deploy virtual images, or libraries of standard images need to be built and made easily available. Users need help with figuring out the right number of resources to ask for and to be able to estimate their associated costs. Costs also should be evaluated not only on an individual application basis but on the scale of an entire project.

At the beginning of this chapter we described three cornerstones of the scientific method: reproducibility, provenance, and sharing. Now we try to reflect on whether these desirable characteristics are more easily reached with Clouds and their associated virtualization technologies. It is possible that reproducibility will be easier to achieve.
through the use of virtual environments. If we package the entire environment, then reusing this setup would make it easier to reproduce the results (provided that the virtual machines reliably can produce the same execution). The issue of provenance is not made any easier with the use of Clouds. Tools are still needed to capture and analyze what happened. It is possible that virtualization will actually make it harder to trace the exact execution environment and its configuration in relation to the host system. Finally, in terms of sharing entire computations, it may be easier to do it with virtualization as all the software, input data, and workflows can be packaged up in one image.

Acknowledgments

This work was funded in part by the National Science Foundation under Cooperative Agreement OCI-0438712 and grant # CCF-0725332. Montage was funded by the National Aeronautics and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology. Montage is maintained by the NASA/IPAC Infrared Science Archive.

References


Cloud Computing: A Viable Option for Enterprise HPC?

Mathias DALHEIMER \textsuperscript{a,1} and Franz-Josef PFREUNDT \textsuperscript{a}

\textsuperscript{a} Fraunhofer Institut für Techno- und Wirtschaftsmathematik, Kaiserslautern, Germany

Abstract. Cloud computing is an upcoming field in distributed systems. Unlike grid computing the most prominent drivers in this field come from industry. The most prominent examples of cloud computing vendors include Amazon, Google and salesforce.com. Their offerings had been developed to solve inhouse problems, such as how to simplify systems management in extremely large environments. Now, they offer their technology as a platform to the public.

While web companies have embraced the new platforms very fast the question remains whether they represent suitable platforms for enterprise HPC. In this chapter we assess the impact of cloud computing technology on enterprises. We present PHASTGrid, an integration platform that allows enterprises to move their workload seamlessly between different internal and external resources. In addition we present our solution for software license management in virtualized environments.

Based on our experience we conclude that cloud computing with its on-demand, pay-as-you-go offerings can help enterprises to solve computationally intense problems. The current nature of the offerings limits the range of problems that can be executed in the cloud, but different offerings will be available in the future. ISVs need to adjust their software licensing model in order to be able to offer their software in these environments.

Keywords. Cloud Computing, High-Performance Computing, Distributed Systems, License Management, PaaS

Introduction

Grid computing has found its place in the scientific community. It is used to allow scientists to run their processing jobs on machines not owned by their own institution, providing better access to computational power. Nevertheless grid computing has not become the unified global compute infrastructure once envisioned. Partially, this is due to the complexity of the infrastructure: while it provides lots of flexibility for scientists it is not well-suited to the requirements of industry users.

Meanwhile, Amazon and others have coined the term “cloud computing” for a different infrastructure that might become the global compute utility. With their “Elastic Compute Cloud”, Amazon offers a simple, metered computing infrastructure that can be accessed by anyone. Other companies have begun to offer similar or extended services.

Cloud computing depends on two enabling technologies: Networks and virtualization. Typically, clouds are accessed over the Internet, regardless of their type. The net-
work connection between the user and the cloud service provider determines the range of applications that can be executed in the cloud in a meaningful way. Today, most homes are equipped with broadband Internet connections. It is therefore an viable option to exchange data in the megabyte to gigabyte range, depending on the application.

Virtualization is the second enabling technology. The advent of products like VMWare or Xen allow the partitioning of commodity hardware. Several operating systems can run concurrently. More important in the context of cloud computing is the fact that the provisioning of new (virtual) machines can be done in software: if the server capacity allows the creation of a new machine, it can do so without any human intervention.

The definition of Jones incorporates this feature as the main characteristic of cloud computing [1]: “Cloud computing is resource provisioning as a service.” Since the provisioning of resources can be handled automatically it becomes feasible to request resources on a true on-demand basis. Combined with the metering of resource usage the cost of IT changes its structure, as we will investigate in this paper. In addition the problem of scalability can be tackled differently: resources can be added and removed as needed.

In order to differentiate the different types of cloud computing offerings consider figure 1. One can distinguish three different types of services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [1]. The different services provide different functionality:

![Figure 1. Comparison of different levels of cloud computing: Infrastructure as a Service provides the lowest level of abstraction, raw hardware can be provisioned on demand. The Platform as a Service adds development tools to the hardware and provides an integrated environment. Finally, Software as a Service provides complete end-user applications.](image-url)
1. With IaaS, one can lease an infrastructure, composed of computing and storage resources. Typically an SLA is included that defines the availability and quality of the resources. IaaS is the virtualized equivalent of a dedicated data center. For example, Amazon offers IaaS: the Simple Storage Service (S3) provides storage on demand. It can be accessed over the Internet. In addition, the Elastic Compute Cloud (EC2) allows users to request the instantiation of virtual machines in one of Amazon’s data centers. Both services are metered: the uptime of EC2 machines and the amount of stored data is billed.

2. IaaS does not provide any additional software to manage the runtime environment – it merely provides the virtualized hardware and operating system. The customer needs to maintain the environment and compensate failures, deal with redundancy and deploy her application. PaaS builds on top of the raw infrastructure by providing a software stack that solves these problems. The platform abstracts from the distributed nature of the cloud infrastructure, incorporating load balancing and availability features. Google AppEngine is an example of such an infrastructure [2]. By using an SDK developers can write web applications that will be deployed in the Google cloud. The runtime environment is completely opaque to the developer – for example, scaling will be handled by the infrastructure without any specialized code in the application. Again, PaaS offerings are metered services: billing is based on the resource usage.

3. Finally, SaaS provides end-user applications on top of the PaaS and IaaS stacks. Based on platform and infrastructure services the application runs “in the cloud”. An example for this are the Google Apps: an office program suite that runs completely on Google’s servers. Scaling and load distribution are handled by Google’s data center technology. Users can simply access the service and do not need to worry about software installation or backups. A similar service is Salesforce.com, a customer relationship management suite.

In this chapter we investigate the applicability of cloud computing techniques for HPC problems in enterprises: are there scenarios where cloud computing offers an alternative to inhouse data centers? First, we define the requirements in section 1. We continue our discussion by presenting our PHASTGrid platform in section 2 and discuss software license issues in section 3. After a discussion of the related work in section 4 we conclude.

1. Applicability of Cloud Computing

The application of IT in enterprise must always provide a value to the business. In the end one can compare the effects of enterprise IT by evaluating the costs caused by and the profits earned by the IT. This is not a simple comparison: for example, the cost of loosing customers because a service was not reachable can only be estimated. There is also the question of technology push: can new IT applications extend the activities of the enterprise in order to create new revenue streams?

In our consulting work we collaborate with companies from various industries in order to improve their IT operation and strategy. Clearly, the promise of instant scalability and on-demand computing are attractive to most businesses. It is, however, not always
possible to move a given application to the cloud. From our experience the following aspects need to be considered:

1. Can the application be executed in the cloud? Are there any technological obstacles?
2. Does cloud computing provide more value for the business?
3. Do legal restrictions prevent the usage of cloud computing resources?
4. Is it possible to integrate the cloud computing resources in the enterprise IT?

These questions result from our discussions with clients. Although other questions might arise we find this list suitable for an initial assessment. We discuss each of the questions in more detail in the remainder of this section.

The question for technological obstacles depend on the type of applications that are used. For example, web applications that are typically hosted in the data center of an ISV can be moved seamlessly to the cloud, mostly without any restrictions. In this chapter we focus on HPC applications. There is a wide range of different HPC application types, not all of them suitable for cloud computing. For example data needs to be shipped to the cloud computing provider in order to be computed. Applications that require lots of data transfers might not benefit from cloud computing. In addition different cloud computing providers offer different runtime environments. Amazon provides virtual machines with no specialized interconnect but connect their machines via Gigabit Ethernet [3]. Walker has compared the performance of Amazon EC2 High-CPU instances with a conventional HPC cluster [4]. The cluster nodes were similar to the EC2 machines except that they were using an Infiniband interconnect. Unsurprisingly the EC2 cluster performed significantly slower than the Infiniband cluster for the NAS parallel benchmark. In contrast, Sun’s network.com provides a Solaris-based compute cluster that also supports MPI over a fast interconnect. We expect that other vendors will start to offer high-performance IaaS in the near future.

The question whether an application can be executed in the cloud needs to be addressed on a per-case basis. In general we expect compute-intensive applications with low data load to perform well in cloud computing environments. A prime example of such an application is financial risk analysis based on the Monte-Carlo method. Another beneficial application area is seismic image processing: although the data load is considerably higher. We present a runtime environment for these two application scenarios in section 2.

Legal restrictions may also hinder the adoption of cloud computing in enterprise IT. If data must not cross country borders cloud computing becomes difficult – in most offerings, it is not clear where the job will be executed. Similarly, data might be too sensitive to be transferred to a data center. Depending on the application there are several possible solutions:

1. Before sending the job to the cloud, all identification data can be stripped from the input data. In the case of seismic data one could remove the GPS coordinates. This way it is not possible to connect the results to real-world locations.
2. Only parts of the processing is done in the cloud – only intermediate results can be compromised. Ideally, the intermediate results do not reveal any useful information to an attacker.
3. Strong encryption might also be used to protect the data. For example the application can decrypt the input data during the computation itself – this would further reduce the exposure of the data.

In addition to these technical measures it is, in our opinion, only a matter of time until providers include suitable SLAs in their offerings such that privacy and legal restrictions are fulfilled.

Cost considerations are another reason for enterprises to choose cloud computing. As Klems et al. point out a careful analysis of the usage of computing resources must be made in order to decide for or against cloud computing [5]. The impact of cloud computing depends on many factors:

1. The cost of running a service inhouse must be compared to outsourcing the service to the cloud. It is, however, not sufficient to compare the alternatives based on measures such as the Total Cost of Ownership (TCO). Assuming that the TCO of the two solutions are similar cloud computing might still be favorable: the cost structure differs significantly. An inhouse deployment requires huge upfront investments in infrastructure and staff. These investments are mostly fixed cost and do not depend on actual resource usage. With cloud computing you do not need to invest in a data center – all costs are variable costs and depend on your resource usage. The capital expenditures can be lowered significantly [6].

2. Utilization becomes an important factor: if a service is not needed it can be shut down without any additional cost. For an inhouse data center this is not possible: the fixed-cost investments have been made when the data center was deployed and must be written off continuously.

3. Opportunity costs must be incorporated as well. For an inhouse data center, overutilization might lead to SLA violations, while underutilization leads to negative financial performance. Typically, the capacity of an inhouse data center is planned according to the expected peak load in order to cope with sudden load increases. With cloud computing, resources can be provisioned on demand in order to handle peak loads. During normal load conditions, the service shrinks to an more economic size.

The opportunity cost example is most pressing for small Internet companies that might experience sudden spikes in their load. The example of Animoto illustrates the volatile nature of load: the small company offers the creation of video clips based on user-supplied images. They run their whole infrastructure using various Amazon webservices. In order to generate the video clips they rely on instances running on EC2. When they experienced a sudden increase in load, they were able to scale from 50-100 EC2 instances to about 3400 instances over the course of five days [7].

In general, one can distinguish several demand behavior patterns: expected demand can be planned in advance because it is known in advance – for example, a web shop might experience increased load in the Christmas season. The Animoto example above is an example for unexpected demand - also known as the “slashdot” effect. For these volatile and unpredictable demand situations cloud computing provides an infrastructure option that allows businesses to scale.

Predictable demand situations arise from regular batch processing tasks. Depending on the size it might be most efficient to use an inhouse data center - the capacity can be planned accordingly. A recent study by the EGEE project found it to be more economical
to build a new data center in order to meet the predicted demand requirements [8]. They assume that they can build an infrastructure that will be utilized very efficiently. In addition the infrastructure will be big enough to lower administrative cost by using automatization. We reason that the findings of this report might not be applicable to other scenarios.

For one-time computations cloud computing might be the most efficient strategy. As Garfinkel points out it does not make a difference whether you use 30 virtual machines for 24 hours or 60 machines for 12 hours [9]. The on-demand nature of cloud computing also saves the cost of buying and deploying an infrastructure. For example, the New York Times used EC2 and S3 to produce free PDFs from their 1851-1922 issues. This did not only save the investment in new machines, it also reduced the time until the task was done because the machines had not to be installed [10].

An unified model for comparing the cost of inhouse data centers vs. cloud computing resources is currently not available. It is foreseeable that cloud computing valuation models will be developed in the future, similar to e.g. Gartner’s total cost of ownership (TCO) models for data centers [11]. A central problem for the development of such a valuation model is how to account for the fast server provisioning and scaling behavior that result from cloud-enabled infrastructures [6].

If the decision to use cloud computing resources has been made, these resources must be integrated in the existing enterprise IT infrastructure. The integration must be as seamless as possible in order to maintain a homogeneous management interface. Ideally, users should not be able to observe a difference between jobs that are executed locally or in the cloud. We have developed cloud computing interfaces for our PHASTGrid middleware, see section 2.

Another issue is the availability of commercial software packages in cloud computing environments: todays software licensing models do not incorporate offsite software usage. This presents a serious limitation for enterprise HPC in the cloud. We discuss this issue in section 3.

2. Service-oriented Computing

Generic Grid middlewares such as Globus [12], Unicore [13] or gLite, which are actively used in the scientific communities, have failed to provide an attractive service oriented computing infrastructure for the industry. Their generic approach that should enable every application to run in a world wide distributed resource infrastructure, owned by various resource providers, has generated a very complex software stack. Standardization of new Grid driven protocols and architectures has become a difficult process with no adoption in the business world.

The obvious success of cloud computing results from the clear separation in infrastructure and application provisioning, the radical reduction of degrees of freedom in the infrastructure and the use of well established standards. It is a similar approach that Fraunhofer took to develop the PHASTGrid middleware.

PHASTGrid was developed in order to make deployment and the execution of applications simpler while focusing on high throughput and low latency. PHASTGrid requires the integration of applications that are then provided as services to the user. This bundling of applications with the platform made the system simpler and guarantees ro-
bust execution of the applications. But the main achievement of PHASTGrid is a internal programming framework supporting the creation of a "job workflow": its main purpose is to accelerate the application by parallelization. Each workflow provides three steps called transform - compute - aggregate and represents a slightly more general programming approach as Google’s MapReduce [14] algorithm. In the end PHASTGrid has become a programming environment for the Cloud. As such it provides a platform as a service and it delivers SaaS.

![Diagram of PHASTGrid infrastructure](image)

**Figure 2.** Service-oriented Computing with the PHASTGrid infrastructure.

Figure 2 represents a view on PHASTGrid from a user perspective. Computing services are offered through webservices and data is uploaded with a data stager based on the WEBDAV protocol. On the infrastructure side we have established a trusted zone that is completely organized by the PHASTGrid job and storage manager. The resource discovery service connects resources that a priori do no know each other, as it is the case within the Amazon cloud. The resource broker is a marketplace for resources and applications and is ready to start auctions between various cloud providers in case they are willing to compete directly with each other.

The acronym PHASTGrid is derived from the main characteristics of the application: automatic Parallelisation, High throughput, high Availability, Service-oriented, fault-Tolerant. In this architecture, three types of agents with clearly defined roles are used: gridservers, jobservers and compute clients, see figure 3.

These agents operate independently, at different levels, inducing a tree-shaped communication graph with the root in a gridserver node. The jobservers are allowed to request jobs from the gridserver and the compute clients are allowed to request jobs from the jobservers. Reversely, the gridserver monitors the jobservers and the jobservers monitor the compute clients. The system is self-healing: if an agent stops its activity or doesn’t react anymore, the monitoring mechanism is able to detect and to correct the situation.
by spawning a new instance of the agent process and by rescheduling the eventually assigned jobs. The gridserver and the jobservers expose a web service interface. On the top of the architecture, a so called gateway service handles the user requests. It is usually deployed into an Apache container and runs in parallel to the gridserver. From a design point of view, all the agents are specialized versions of a generic prototype, performing similar operations but in different ways. Each agent has its own job manager, slave manager and event monitoring manager:

1. The job manager maintains different job queues and states of the jobs.
2. The slave manager implements the bookkeeping of the slaves and of their assigned jobs.
3. The event monitoring manager performs periodic checks of the application’s integrity, prepares statistical reports about jobs and agents and undertakes fault-tolerance measures, if necessary.

One of the main characteristics of PHASTGrid is the ability to treat jobs in a uniform way, regardless of their type. We call the implementation of a job on PHASTGrid a workflow. New job workflows can be written without detailed knowledge of application’s internals. This mainly consists in implementing a hierarchy of classes derived from a generic job class and overriding certain methods, where necessary. These methods mainly concern the way how the job is decomposed into sub jobs, by either taking into account a simpler or a more complex performance prediction model, the job is computed, the results are aggregated into a higher level result, and how the job data is transported. The decomposition into sub jobs can be implemented adaptively, depending on the jobtype’s requirements.
We will now present examples and explain how to extends the local computing capacities into the cloud. We focus on two examples: in the financial industry, Monte Carlo simulations are used extensively in risk management. In the seismic industry various image processing algorithms are used to process huge datasets.

2.1. Monte Carlo Simulations in the Financial Industry

PHASTGrid is used in the financial services industry since 2004. Monte Carlo simulation is the main compute task in the equity derivative market simulation. As an example we use the evaluation of a portfolio of arithmetic average Asian options. We started with a code supplied by our financial mathematics department. The code was designed to run on an Opteron machine and used in production. The initial runtime was 21.9 seconds, see table 1. We reimplemented the code using the Mersenne Twister and Box-Muller algorithms for random number generation. In addition we use the SIMD instruction set. The runtime decreased to 3.5 seconds on a single Intel Core.

We subsequently ported the optimized application code to the Cell platform where we used only one of the available eight SPEs. The runtime decreased to 1.1 seconds. So far we did not put any effort in distributing the job in the cloud, but this example shows that the optimization on a single-core level is usually a good idea.

In the next step we integrated the application in PHASTGrid. We developed an application-specific webservice that takes the user-supplied parameters and triggers the internal execution of the job. We also implemented a new workflow in order to handle the job. During the transform step we parallelize the execution of a single Monte Carlo job into several sub jobs: we split each job based on the requested number of Monte Carlo iterations. For example, if a request asks for 100,000 iterations, we might split it into 10 chunks of 10,000 iterations each. The exact number of the chunks depends on the request parameters and is decided dynamically based on a speedup model. After the jobs have been computed during the compute steps, we aggregate the result and sent it back to the user.

In order to evaluate the performance we conducted a series of experiments on both local and cloud resources using the optimized financial Monte-Carlo code. We submit portfolios of 20 Asian options sequentially and measure the turnaround times, see figure 4. Each evaluation of an Asian option triggers 4 independent jobs, so each portfolio evaluation consists of 80 jobs. We repeat the experiment 100 times.

Our local resource is a QS22-based Cell system where we use 10 CPUs, each of which has 8 SPEs where the calculations run. The mean turnaround time is 3.939 seconds for the whole portfolio. The confidence interval is [3.715, 3.924]. We see the same qualitative behavior for our runs at Amazon’s EC2 service. We use 10 instances of the “Large” type, featuring 2 virtual cores running 64-bit Linux. The mean turnaround time is 9.254 seconds with a confidence interval of [8.754, 9.625].

The performance of the QS22 system is significantly higher than the corresponding system at Amazon. This is mainly a consequence of the different CPU types since we have on the Cell System 4 times the number of cores. Due to the overlap of computing and data transfers we only see a marginal effect from the network latency although we are using the US based data centers of Amazon. The variance of the turnaround times is 0.012 for our local Cell installation and 0.326 for the Amazon deployment. Although noticeable, the variance of runtimes is acceptable for this application.
Figure 4. Boxplots of the turnaround times of our experiment. We compare the behavior of our Monte-Carlo Asian Option evaluation code on a local Cell cluster (QS22) with instances running at Amazon. The X-Axis is logarithmically scaled for better readability.

Table 1. Comparison of the performance of different single-core implementations. The averaged runtimes for calculating one million paths for a fixed set of parameters are shown. The CBE outperforms the Intel architecture.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Average runtime for 1 million paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original single-core Opteron code</td>
<td>21.9 sec</td>
</tr>
<tr>
<td>Intel single-core Woodcrest</td>
<td>3.5 sec</td>
</tr>
<tr>
<td>Single CBE SPE</td>
<td>1.1 sec</td>
</tr>
</tbody>
</table>

To expand the local computing capacities to the Amazon cloud we have two options in our scenario. On one hand, PHASTGrid can expose two different web services to the user: one is launching the applications in the cloud while the second is using the local resources. For a quick startup PHASTGrid will keep a jobserver running at the Amazon’s EC2, which generates only marginal costs. The number of compute resources can be adjusted dynamically. While feasible this option burdens the user with the adequate choice of internal and external resources. This might not be desirable.

The other option lets PHASTGrid use both local and external resources concurrently. We would deploy a local jobserver and a remote jobserver at the Amazon’s EC2. This enables us to hide the internal resource management from the users. In addition we can change our compute capacity depending on the workload: If the local resources become overutilized we move jobs to our cloud resources. The pool size of the cloud resources can be adjusted as well.

2.2. Seismic Image Processing

As a second viable use case we have evaluated seismic data processing. PreStackPro is a commercial product used to visualize and process seismic data. A typical installation is running on a dual workstation setup. In the seismic processing world we have two different compute operations: On one hand small computing tasks like filter operations that are used to improve the data quality. On the other hand very compute intensive tasks are used, for example wave equation based shot migration. The filter operations can run on the workstation setup without any problem in a few hours on full a dataset, which is about 1 TB in size. For the compute intensive tasks like shot migration the processing of a typical dataset (100 GB) will take half a year.
We have integrated a PHASTGrid client in PreStackPro in order to control the compute tasks from the GUI directly. This way, the user can continue to work on the data while the compute intensive tasks are offloaded to a different resource pool.

Moving the compute intensive tasks to an IaaS provider is an interesting alternative because the processing times can be lowered significantly. The computing time for a single shot on a large Amazon EC2 instance is about one hour while the input data size is about 10 MB. Data transfer times and cost are negligible. PHASTGrid can handle the load and data management. This allows us to provision a few hundred EC2 instances in order to reduce the processing time to a day.

The last example is the key to the understanding of the impact of cloud computing. Small companies that will never buy a large cluster are enabled to compete and scale their business. PHASTGrid is an example of a middleware that provides the integration platform as a service and wraps application software in easy to use services. The PHASTGrid enabled parallel programming approach is quit general and transforms the cloud in a productive parallel computing environment.

2.3. Experiences

During our experiments with the usecases described above we found cloud computing to be an interesting alternative to inhouse data centers. We don’t expect that all applications are suitable for cloud computing environments. But for applications that need to compute for a long time on a small dataset the use of cloud computing facilities has proven to be an option.

As the example of seismic image processing demonstrates it is feasible to acquire massive amounts of resources in order to decrease the processing time significantly. Since the resource usage is metered an enterprise only needs to pay for the real resource usage. Compared to inhouse data centers this can provide a cost-effective solution.

Another usage scenario are overspill capacities: if the local data center cannot handle the load additional resources in the cloud are provisioned. A seamless integration of the external resources is needed, as described in section 2.1. PHASTGrid can deliver this integration, see figure 5. With its webservice-based job submission interface one can implement various scenarios. The location of the compute facilities is not relevant to PHASTGrid: inhouse data centers and cloud computing providers offer the IaaS layer. PHASTGrid bridges the different IaaS offerings and delivers the application to the user. The resource pool can be adjusted dynamically as the load changes.

There is, however, a drawback to using IaaS: since the processing is done at an external site, data security needs to be considered. We see three ways of dealing with this issue:

1. First, the data can be encrypted before shipping. The application itself loads the encrypted data and decrypts it in memory. The results are then encrypted again before shipping them back to the user. An attacker would need to read the data from memory during runtime.
2. All identifying data can be stripped before submitting it to external resources. In the case of seismic image processing one could remove the geographic location from the dataset – rendering the information useless to an attacker.
3. Only parts of a processing chain could be sent to IaaS providers. This way an attacker could only observer intermediate data. When additional noise data is
PHASTGrid provides an abstraction over different IaaS providers. It incorporates the application in its runtime management.

included in the datasets it might not be possible to identify the result of the whole processing chain.

More research on this topic is needed. Another obstacle for cloud computing is the issue of license management which we will discuss in the next section.

3. License Management

Software license management is an important issue for independent software vendors (ISVs). Typically, they grant licensees the right to use their software by a licensing contract. The contract defines the usage rights that are granted by the licensor and the fees that need to be paid by the licensee. Obviously, the management of the licenses is the core of the business model of any ISV. It is therefore important to recognize the legal, business and technical requirements of license management.

In this section we present our work on GenLM, a license management technology that is suitable for grid and cloud computing environments. We start with a short introduction of the economics of software licensing before we present the technology that addresses these issues.

3.1. Economic and Legal Aspects of Software Licensing

The business perspective of ISVs changes drastically in cloud computing. Each ISV has a different pricing model and the underlying business models can differ significantly.
From discussions with different ISVs we extracted a simple model – further examination is needed in order to assess more complex models.

An ISV develops the software before any income can be made by selling it. The problem of determining the price for a software license is complex, different techniques can be used. In our simple model, the ISV calculates prices based on the number of licensees and the intended income. Issues such as whether a given price is attractive for users etc. are not incorporated. We also assume that all users can get a similar price. The income of the licensor can then be determined by the following equation:

Please note that we assume that each user is an end user of the software. This changes in grid or cloud computing environments: users can access resources at other sites. If the site owns a license, the remote user might also be allowed to use the license. In fact, there is typically no way to differentiate between local and remote users – licenses are used on a first-come first-served basis. From a licensor point of view this is a problem: since the grid user doesn’t need to buy a license on her own the number of users and therefore the licensors income decreases. Consequently, ISVs ask for higher prices when offering licenses to grid and cloud providers.

From a legal point of view the license usage of grid users is typically not covered by the license agreement. The contracts refer only the licensee and typically prohibit the use of the license by third parties such as a grid user. This leads to a problem for resource providers: while it might be reasonable to offer grid licenses of certain software packages there are no license agreements that allow this kind of usage.

In our work in the German grid project “PartnerGrid” we observed these issues. We collaborate closely with two ISVs in order to develop techniques to overcome these problems while addressing the concerns of ISVs at the same time. The requirements can be summarized as follows:

1. **Support for existing license agreements**: in practice it is almost impossible to change legal contracts during their lifetime. A licensing framework must support existing contracts.

2. **On-Demand licenses**: in addition to the classical “CPU-per-Year” licenses, it should also be possible for an ISV to offer on-demand licenses when desired. In combination with the first requirement this would allow complex licensing terms: for example, an user must purchase a base license which is valid for a year. Additional software modules could then be licensed on-demand on top of the existing base license.

3. **Mobility of licenses**: Typically, grid providers and cloud computing data centers are usually not involved in the license agreement. Technically, the license should be valid regardless of the execution location of the computations.

Beside these requirements we need also to fulfill certain non-functional requirements:

1. The framework must be easy to integrate with the existing software packages. Ideally, the existing license management routines will be replaced or enhanced by the new framework. The framework must also be portable across different operating systems.

2. Support for both grid and cloud computing environments must be built in. This requires special care regarding network friendliness and security.

3. The system must be highly available. If users can’t request licenses when they demand them, this might have serious implications on the ISV.
4. The system must be secure. Licenses must not be granted without a license contract. Additionally, it must be hard for malicious crackers to break the license checks in the software itself.

Before we present our technique in section 3.3 we discuss the currently available products in the next section.

3.2. Shortcomings of Available License Management Techniques

In order to be suitable for grid environments a license management technique must allow the use of software licenses in virtualized environments. Currently, software license management products fall into three categories:

1. **Hardware Tokens**: During startup, the application checks whether a hardware token is present. Since the token needs to be plugged in the real hardware this is not a suitable technique for virtualized environments.

2. **System Hardware Fingerprinting**: During installation of the application a fingerprint of the host hardware is generated. This fingerprint includes information such as the CPU ID, MAC addresses etc. The fingerprint is then sent to the ISV which generates a corresponding key. After the key is entered the application is unlocked on the current hardware. Again, this technique is not suitable for virtualized environments since the hardware is usually not known in advance.

3. **Network Licenses**: The software license management is implemented in a server that is deployed in the customer’s LAN. The application polls a license from the server during startup and releases it during shutdown. In general this seems to be an applicable solution to license management.

A possible usage scenario for a networked license manager is the following: in addition to hardware resources the resource provider offers software licenses as well. A user requests the execution of a commercial application which in turn relies on a network license server. The resource provider bills its customers for the license usage. Although this is feasible in general this configuration is currently not possible due to legal restrictions: as outlined above, ISVs do not allow the usage of licenses by third parties (which would be the user in this case).

Another problem arises from the implementation of current license management products. The market is dominated by Aresso’s Flexnet [15]. Currently it only supports an IP-based authentication scheme: All clients requesting licenses from within a certain subnet are allowed to request licenses. It is therefore difficult for the provider to link license usage to specific customers or to implement different pricing schemes. The BE-inGRID project addresses these issues by implementing a proxy between the application and the license server.

The license could also be requested from the user’s license server. While this usage is legally feasible the license server would need to be exposed to the Internet. The current authentication schemes would allow any outside user to request licenses from the license server – which is certainly not in the interest of the user.

3.3. GenLM: An Implementation of Mobile Licenses

In order to address these shortcomings we propose the concept of mobile licenses which are independent of the location where the application is started. Users, resource providers
and ISVs form the three stakeholders of grid computing. We have developed one compo-

tent for each of the stakeholders, see figure 6. The GenLM client is used by the user or by a preprocessing software to acquire a license token. The GenLM server is responsible for issuing licenses on demand. The GenLM license verifier is included in the ISVs compute software and checks whether a license is valid for the pending job.

The central idea of GenLM is to attach the software license to the input data of the batch job. We create a license token for a given set of input data. The license token is a file that can be transferred together with the input data to the compute site. It contains all information the license verifier needs in order to check the validity of the software license.

In order to clarify the concept of our license token we will shortly outline the lifecy-
cle of the token. A token consists of a set of hashes of the input files and a license terms specification which is software-specific. The token is generated by the GenLM client and signed by the GenLM server. During job startup the token is evaluated again, see figure 6.

When a user wants to submit a job she needs to acquire a valid license token for her input data. The GenLM client starts by computing a cryptographic hash for all input files. These hashes are stored in the token. By the construction of the hash they uniquely identify the input files. In addition the license terms requested by the user are stored in the token. These license terms are ISV-specific and would typically contain information such as the number of requested cores, the software modules to be used for the job etc. GenLM doesn’t evaluate these terms – typically, this information is used by the ISV to decide which license to issue for this specific job.

The request token is then signed with the user’s X.509 certificate and sent to the GenLM server. The server extracts the license terms and the user’s identity from the request token. This information is forwarded to a policy plugin which can be implemented by the ISV. The purpose of the plugin is to enforce the ISVs business model. For example the user’s identity can be matched against a customer database. Depending on the contract of the user’s organization the request will be billed separately or it is covered by a flat-fee license agreement. All necessary steps in order to be able to bill the customer will be made in the policy plugin. This might involve putting a billing record in a database.

Assuming the license request is granted the server uses its own certificate to sign the request token. It is then sent back to the user. We call the signed request token a license token since it contains a valid license.

Together with the input data the license token is transferred to the compute site. A job is enqueued at the site which will finally compute the results. On job startup the

Figure 6. An overview of the GenLM components. The figure shows the logical message exchange between user, ISV and resource provider. Please note that the message queue and firewalls are not shown.
license token is inspected: First, the signature of the token is inspected. The public key of the license server is used to verify the signature of the license token. If the signature is correct, the application computes the input file hashes based on the local files and compares them with the hashes stored in the token. Given that the locally computed hashes are identical to the hashes stored in the license token, we know that the license server granted this job. The computation can start.

As outlined above we rely on hashing and signature algorithms. We sketch our approach subsequently. Let $I$ be the set of input files. The request token is a tuple $RT = (LT, H_I)$ with license terms $LT$ and a set of hashes of the input files $H_I$. The hashes are generated with a collision-resistant one-way hash function $h$. For all input files $i \in I$ we compute $H_i = h(i)$. By construction of the hash function we get a practically unique identification of the contents of file $i$.

When the request token is signed by the GenLM server we rely on asymmetric encryption. The server has a key-pair $(p, s)$ where $p$ is the public key and $s$ is the private key. The key-pair $(p, s)$ must satisfy the requirements of the encryption function $e$. In order to sign the request token the server uses its secret key to encrypt the token $T$:

$$T = e_s(RT)$$ (1)

This signature is then attached to the original token. When the license verifier evaluates a license token it uses the public key $p$ of the server to reconstruct the token that was signed $(ST)$

$$ST = e_p(T)$$ (2)

If $ST = T$ we can guarantee that the token was signed with the server’s private key – otherwise, we reject the job.

For our implementation we rely on algorithms recommended by the German “Bundesamt für Sicherheit in der Informationstechnik” [16]. The RSA and SHA-256 algorithms are considered safe up until 2013. We choose the OpenSSL implementation because it is widely available and maintained by a huge user community [17].

The implementation itself is based on C++ and currently in a closed beta test. It will be made available as a commercial product for Windows and Linux platforms. The software architecture is optimized for modern multicore machines. We adopted the SEDA architectural approach as proposed by Welsh [18]. In order to be able to implement different licensing schemes we provide the possibility to integrate arbitrary policy engines written in C++ and Ruby. Please note that we patented this solution.

4. Related Work

Cloud computing is a new field that is currently in its infancy. In addition, most of the current work is conducted in an industrial context. These two factors can explain the lack of extensive scientific literature on this topic.

There are a variety of works available that introduce the current state of cloud computing and provide an overview. Jones presents the landscape of cloud computing services [1]. Weiss provides another introduction to cloud computing [19]. Vogels presents the technical and economic reasoning behind Amazon’s IaaS offerings [6].
In the field of performance evaluation there is a lack of work. The papers by Walker [4] and Rightscale.com [3] are first steps toward a better understanding of the performance behavior of compute clouds. Further research might include the impact of virtualization on the performance. The issue of network availability is also not addressed. All these evaluations are, however, highly dependent on the provider of the cloud computing services.

The question of cost effectiveness of cloud computing infrastructures is currently investigated by Klems et al.[5] and Deelman et al.[20]. Klems attempt to build a framework for estimating the economic impact of cloud computing vs. inhouse infrastructures. They build their view on the concept of opportunity cost. This allows them to incorporate the improved reactivity of cloud computing solutions – if a service experiences demand spikes, the TCO can be reduced since the amount of resources can match the demand. This leads, however, to increased complexity of the model compared to other models such as TCO[11].

Deelman et al. analyse the cost structure of running scientific applications in the cloud. They consider the cost of data transfers, storage resources and compute resources in several scenarios. Depending on the size of the data and the computation cost Amazon’s IaaS service can lower the operational cost of their astronomy application [20].

Altmann et al. have presented a taxonomy of grid business models which can be considered when building a similar taxonomy for cloud computing [21].

A number of studies deal with the question whether scientific computations can be made on Amazon’s IaaS offering. Hazelhurst [22] concludes that EC2 will not replace local HPC clusters but complement them for certain applications. Garfinkel [9] also presented his findings of using Amazon’s service in his research. CERN considered using Amazon services when the need for more compute capacity arised [23][8]. Other studies include Evangelinos et al. [24] and Rehr et al. [25].

From a technical point of view the field of cloud computing presents challenges with regard to reliability and middleware design. Reliability is often implemented using peer-to-peer technologies such as Amazon’s Dynamo [26]. Vogels also discusses the relaxation of consistency in order to provide high availability [27].

Generic grid middlewares such as Globus [12] and Unicore [13] share the vision of a global computing infrastructure. They introduce a layer on top of local resource management systems and map local services to services that can be accessed from outside the organization. Some commercial software providers also deliver grid middleware solutions. DataSynapase targets the financial services industry with their service-oriented middleware [28]. Platform Computing provides another solution for distributed job management [29]. These middlewares extend the existing HPC infrastructures and expose local cluster resources and filesystems. They are, however, not based on the idea of low-level resource virtualization. Compared to PHASTGrid they offer more flexibility at the price of a more complex infrastructure.

Several distributed platform designs have been presented earlier, including Google’s MapReduce implementation [14]. Yahoo’s Pig Latin was described by Olson et al. [30] [31]. Both techniques manage the distribution of work in big clustered systems, similar to our PHASTGrid technology. They also incorporate facilities that deal with failure and runtime optimization.

Currently, two European projects develop licensing solutions that target virtualized environments. The BEinGRID project provides a license tunneling mechanism for
FLEXnet-based products [32]. The SmartLM project [33] builds a software management solution based on WS-Agreement [34]. A license contract can be (re-)negotiated during various stages of the job lifecycle.

Li et al. reviewed the requirements for license management in grid and cloud computing environments [35]. They argue that licenses should be managed in terms of a service level agreement and scheduled by a scheduler. We disagree because it is not in the interest of the ISVs to have their resource usage optimized by a scheduler. There is no incentive for ISVs to follow such an approach.

A variety of commercial license management products is available on the market. The most common known include Acresso’s FLEXnet [15], IBM’s LUM [36], HP’s iFOR/LS [37] and TeamEDA’s License Asset Manager [38]. All license management software products on the market do not reveal their inner workings. It is therefore hard to compare our approach to these solutions. To the best of our knowledge, all these products manage licenses on a node-lock or floating license as described in section 3.2.

5. Conclusions

Cloud computing has the potential to replace certain inhouse resources in industry. For web applications such as Animoto [7], the currently available offering of Amazon represents an interesting alternative to traditional inhouse data centers. While certainly usable for some applications the Amazon offering is not suitable for the classical HPC use case, mainly due to the fact that fast interconnects are missing [4]. But we expect that adequate offerings will evolve in the foreseeable future. We also expect that enterprises will choose to utilize different IaaS providers in order to optimize their IT operations - as the comparison between x86 and Cell architectures for the portfolio evaluation showed.

Returning to our questions from section 1 we give the following answers:

1. *Can the application be executed in the cloud?* This is clearly dependent on the application. Unsurprisingly we argue that applications with a high computation to communication ratio can benefit from the cloud computing approach. In addition the data transfers to and from the cloud computing resources need to be considered [20].

2. *Does cloud computing provide more value for the business?* In terms of flexibility and scalability cloud computing has a lot to offer. Here we see the main challenge in providing a platform for the management of the application in the cloud. Our PHASTGrid middleware provides such a platform.

3. *Do legal restrictions prevent the usage of cloud computing resources?* Here we identify two barriers: on one hand, if the input data must not leave the organization it is not feasible to use cloud computing. On the other hand, if the application is commercially licensed, the current licensing technology does not account for virtualized environments. Our GenLM approach can help ISVs to offer licenses that work in cloud computing.

4. *Is it possible to integrate the cloud computing resources in the enterprise IT?* Cloud computing resources can augment the inhouse data centers of enterprises. Using the SoC approach our PHASTGrid middleware provides a basis for enterprise cloud integration.
Our service-oriented computing approach helps to abstract from the different implementations of IaaS providers. We can provide a dynamic environment where a reliable platform abstracts from the underlying implementation. It is possible to choose the adequate infrastructure on a per-job basis. Different applications can coexist, allowing to consolidate the infrastructure. It is also possible to mix inhouse and external resources in order to improve the cost-effectiveness.

Software vendors need to address cloud computing by introducing new license management techniques. When enterprises move computational facilities to cloud service providers, the business model of ISVs must account for this. We have introduced GenLM, an implementation of mobile licenses. The idea of licenses that can travel with the job allows users to select the best licensing option. At the same time, ISVs can decide on the licensing terms for each user independently. The resource provider does not need to deal with different license agreements for different users.

Acknowledgements

The Fraunhofer ITWM supported this work. We would like to thank Alexander Petry and Kai Krüger for valuable discussions and support. The development of GenLM was supported by the German “Bundesministerium für Bildung und Forschung” under contract no. 01G07009A-D in the PartnerGrid project.

References

(2008) A lyso software-hardware.htm


Evidence for a Cost Effective Cloud Computing Implementation Based Upon the NC State Virtual Computing Laboratory Model

Patrick DREHER\textsuperscript{a,1}, Mladen A. VOUK\textsuperscript{b}, Eric SILLS\textsuperscript{b}, Sam AVERITT\textsuperscript{b}
\textsuperscript{a}Renaissance Computing Institute, Chapel Hill, NC 27517, USA
\textsuperscript{b}North Carolina State University, Raleigh, NC 27695, USA

Abstract. Interest in cloud computing has grown significantly over the past few years both in the commercial and non-profit sectors. In the commercial sector, various companies have advanced economic arguments for the installation of cloud computing systems to service their clients’ needs. This paper focuses on non-profit educational institutions and analyzes some operational data from the Virtual Computing Laboratory (VCL) at NC State University from the past several years. The preliminary analysis from the VCL suggests a model for designing and configuring a cloud computing system to serve both the educational and research missions of the university in a very economical cost efficient manner.

Keywords. Cloud Computing, VCL, Cost Effective

Introduction

The concept of cloud computing has become a popular phrase in information technology (IT) over the past several years. Cloud computing can be defined as a computational paradigm that, at the requested and appropriate level, seamlessly and securely provisions a wide range of IT services through a combination of connections, computers, storage, software and services accessed over a network. A well designed cloud is based on a service-oriented architecture, and is capable of providing a rich set of customizable services.

Cloud computing has several key characteristics that provide users with a unique capability and niche among computational systems. Clouds can provide device independence from any particular hardware vendor, and offer implementation of resource and cost sharing from among a large pool of users. Within this resource sharing concept, specific implementations help to enhance these general gains in technical performance, with potential follow-on economic savings. For example, technical efficiency and scalability are enhanced through relative centralization of infrastructure, with location and device independence, and with efficiency in utilization.

\textsuperscript{1}Corresponding Author: Patrick Dreher, 100 Europa Drive Suite 540, Chapel Hill, North Carolina 27517, USA, dreher@renci.org
through management of user demand loads to a cloud system component via implementation of software that controls simultaneous multi-user or project access.

Beyond these general technical enhancements, individual cloud architecture designs, specific implementations, and usage profiles have the potential for additional technical and economic impacts that can lead to better performance, throughput, and reduced costs. Areas at each specific site where such economies may be improved include:

- Network bandwidth and network load to the system
- Reliability and availability (“up-time”) of the system
- Site specific operational profile, including concurrent resource usage profile
- Services mix (IaaS, PaaS, SaaS, and AaaS - which ones and in what proportion)
- Efficient on demand allocation and aggregation, and de-allocation and de-aggregation, of Central Processing Unit (CPU), storage, and network resources
- Type of virtualization used (bare-metal to virtual machine ratio)
- Scalability and rate of adaptability of the cloud to meet changing user demands
- Sustainability of the system under varying workloads and infrastructure pressures
- Serviceability and maintainability of the architecture along with the overall cloud computing system and user interfaces and application programming interfaces (API)

One such implementation that incorporates these user requirements and design specifications is the Virtual Computing Laboratory (VCL) at North Carolina State University. VCL is an award-winning open source implementation of a secure production-level on-demand utility computing and services oriented technology for wide-area access to solutions based on real and virtualized resources, including computational, storage, network and software resources.

VCL differs from other cloud computing implementations in that it offers capabilities that are very flexible and diverse ranging from Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Applications as a Service (AaaS) options. These capabilities and functionalities can be combined and offered as individual and group IT services, including High-Performance Computing (HPC) services. VCL is also open source [12], and highly modularized so that a knowledgeable end-user can replace components, and not locked into a particular IaaS, PaaS, SaaS, AaaS or other component or solution. The HPC service is a very important feature of VCL. The system very successfully integrates HPC into cloud computing by managing not only resource capabilities, but also providing efficient coupling among the resources.

---

2 Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), e.g., http://www.webuild.org/2008/07/cloud-computing-basics.php, http://paastalk.com/cloud-saas-pass-market-overview/, http://www.webcloudworld.com/analysis/a-map-of-saas-paas-cloud-computing-players/. Application as a Services (AaaS) can be an even higher abstraction where the end-user is only interested in the general application functions regardless of software that is providing it, platform it is running on, or infrastructure. For example, a document reading function - “I would like to read the document sent to me, but I do not want to know or care whether it is in Word, PDF, text, WordPerfect, etc..”
We recognize that commercial organizations have also been aggressively building cloud computing capabilities with claims of economic advantages for their new architectures and operations methodology [e.g., 1, 2 and references therein]. These types of statements have sparked a vigorous debate with arguments both in favor and against the economic viability of cloud computing. [e.g., 3]. We will leave aside a discussion regarding the economic pros and cons for a commercial cloud computing operation. In this paper we will focus specifically on a cloud computing implementation within a research-oriented educational institution of higher learning, and discuss some of the factors that demonstrate how such a system provides a scalable, sustainable, economically valuable, and viable contribution to the campus layer IT CyberInfrastructure. Comparisons among educational and commercial cloud computing implementations will be explored in a subsequent publication.

1. Building an Economically Viable Model for Cloud Computing

1.1. User Requirements

Cloud computing systems serving users within a university environment must at least provide the following capabilities for the faculty, researchers, staff and students

- Reliably deliver services and support to a wide range of users from the novice to the most sophisticated expert researcher - from users who can barely find the terminal, to those who are expert supercomputer users.
- Reliably deliver a wide-range of course materials and academic support tools to instructors, teachers, professors, and other educators and university staff as part of the academic mission of the institution
- Reliably deliver research level computational systems and services in support of the research mission of the university

Fulfilling such a set of user requirements across components of a distributed system of hardware and software, that is also often coupled with a given level of network support, is sometimes categorized under the term of a “service–oriented architecture”. These types of IT systems provide the end-user with a given functionality, capacity, and quality of delivery connected through a mix of some combination of both tightly and loosely coupled components.

The distributed components have characteristics with qualities that that can be described as

- On-demand or batch
- Reusable
- Sustainable
- Scalable
- Customizable
- Secure
- Reliable and fault-tolerant

In addition to all of these technical characteristics, a properly designed system must also have data and process aware service based delivery and the ability to audit processes, data, and results for both individual and workflow-based services. Finally the IT designers of such service oriented architectures must demonstrate that these systems are cost-effective to operate and maintain in an educational environment.
From this list of requirements, proposed hardware and software architectures with operational profiles are developed that will support the users’ education and research work [4, 5]. These proposed designs are based on the generic assumptions that in a typical environment at an institution of higher education the various levels of users include

- Service end-users (e.g., students in a class)
- Service integrators and extended content creators (e.g., faculty, teaching assistants)
- Basic-services developers
- System experts.

End-users are typically interested in having a system which is user friendly, flexible, and responsive to their needs. Most end-users view a computing system as a tool to assist and enhance their educational activities, and/or support their research efforts. The service integrator is assumed to have a higher level of IT knowledge with respect to the system and utilizes this skill to prepare and load educational or research material onto the system for access by the end users. The developers provide both the base-line computing infrastructure and develop services in support of the system. In many instances, developer may also be researchers. Finally, the system experts handle the advanced system features, enhancements, and new functionalities.

By far, the largest fraction of users in a higher educational institution are the student and faculty end-users. This group for most part is the driving engine for such CyberInfrastructure capabilities and whose usage patterns can validate and justify the economic aspects of any cloud computing or infrastructure service implementation.

1.2. The Virtual Computing Laboratory Design

The VCL design and architecture began with the premise of a secure, scalable, maintainable, and sustainable service-oriented architecture (SOA). This system design would be needed to deliver user required solutions for a variety of diverse service environments, anytime and anyplace, on demand, or by reservation. The system users needed to be able to configure single real or virtual computer laboratory “seats” or desktops, single applications on-demand, classroom size groups of seats, enterprise server solutions, implement research clusters for specific calculations, deploy aggregates of resources to deliver sub-cloud service, and high performance computing services. Figure 1 illustrates the spectrum of VCL service categories that we found in demand in a university environment.

To address these user requirements, Sam Averitt et al. [7] designed and developed a technology called Virtual Computing Laboratory (VCL). The effort was a collaboration between the NC State College of Engineering and the Office of Information Technology [e.g., Vouk08a]. The basic VCL design consists of four principal architectural components

- End-user access interface
- Image repository
- Resource-manager
- Computational storage and networking hardware

3 At the time called Information Technology Division
Most end-users access VCL through a web-based interface. However, the same VCL functions can be accessed through a network-based API. Depending on the user access permissions, there are several layers of functionalities that different user categories can access. A lowest access level allows the user to select from among a suite of operating system and application combinations, called images. End users select from a variety of images that have been stored in an image repository constructed for a wide variety of disciplines. A user is allowed to make a reservation of these resources for immediate use, or for later use. Basic users are provided with only a limited number of additional service management functions. For example, basic users can extend their reservations and look at VCL statistics.

Within VCL, an “image” is defined as a software stack that incorporates:
- Any base-line operating system, and if virtualization is needed for scalability, a hypervisor layer
- Any desired middleware or application that runs on that operating system
- Any end-user access solution that is appropriate

Depending on how they have been constructed, images can load to “bare-metal” hardware or to a hypervisor. More advanced users can save the current state of the images they have reserved, can add or delete applications, and select other advanced features.

The VCL resource manager consists of two principal parts. There is an image loader and platform manager, and a resource and image scheduler. The manager maps the user’s request onto available software application images and available hardware resources (including heterogeneous hardware platforms). In addition to the basic scheduler, the VCL Manager includes security capabilities, multi-site coordination, performance monitoring, virtual network management, and reporting utilities. This allows users to check their current reservations in the system, set some level of preferences, view system statistics, and access help files and support.
A user can have either sole use of one or more hardware units, if that is desired, or the user can share the resources with other users. Scalability is achieved through a combination of multi-user service hosting, application virtualization, and both time and CPU multiplexing and load balancing.

VCL architectural design offers a wide spectrum of services ranging from single real or virtual computer laboratory “seats” or desktops, to single applications on-demand, to classroom size groups of seats, to enterprise server solutions, to homogeneous and non-homogeneous server clusters for research and production, to high-performance computing services and clusters (including grid-based services). An advanced user can construct one’s own services by building and storing customized VCL images, including aggregates of two or more VCL images (so called composite images), thereby extending the service capabilities of the system. Additional advanced management functionalities include

- Creation of extended reservations (by date) – typical student reservations are in the range 1 to 4 hours, however long term reservations may be needed for HPC service, continuous services, research projects, etc.
- Creation of block reservations (e.g., 25 one hour classroom seats at 11am every Monday)
- Management of user groups
- Management of image groups
- Management of schedules for the resources used by images
- Management of and grouping of resources
- Management of VCL management nodes
- Viewing of resource time-tables
- Setting of privileges for users
- Identification of users

2. Operating VCL as an Economically Viable Model for Cloud Computing

Today both educational institutions and commercial vendors are deploying cloud computing resources and services, each with a somewhat different emphasis. On the commercial side, companies such as Amazon, Microsoft, Google and others have entered this area, each offering users a different mix of capabilities. For example, Amazon provides virtual (and exceptionally physical) hardware, and user controllable kernels and software stacks. On the other hand, companies such as Force.com offer cloud resources that run against a very constrained set of applications. Depending on the commercial operation, users may have flexibility in assembling the cloud hardware but lack the additional systems to support such type of configurations without additional development, or they may be highly constrained to run only specific types of applications [2].

An environment in an educational institution often has a complex set of operational requirements that may include openess, accessibility, mobility in transferring project information to and from the cloud system, control of the configuration of the hardware and software stacks, a capability for a richness and flexibility, and a level of security of their computations, data and intellectual content [1, 10]. In addition, there are also capital equipment and operational considerations. Such
a list of requirements may be difficult to fulfill in the commercial world at the price-point acceptable to educational institution.

When constructing a cloud computing system, there is a delicate balance between acquiring too many computing resources that are not efficiently utilized throughout the year and having an insufficient quantity to satisfy user demand during periods of maximum load. Having an excess of capital resources on the chance that they may be needed during peak use periods, may result in long periods of time where these resources are idle and being wasted (e.g., during summer). At the other end of the supply-demand spectrum, under-provisioning of a cloud computing system can lead to serious dissatisfaction among users who do not receive the service they desire at the time they request it.

Both scenarios can be inefficient and each incurs a different economic cost. In the first case are the issues that arise from underutilization. In the second scenario, a scarcity of resources leads to users who not serviced and thus are dissatisfied.

2.1. VCL Operations

VCL powers the NC State cloud. Users are validated and authenticated into the VCL system using a variety of methods, including LDAP and Shibboleth. Authorization to check availability, and schedule VCL resources and image installation onto the cloud resources, are controlled by one or more distributed management nodes. Using image-associated meta-data, VCL checks that licensing, and any other constraints, are honored when resources are scheduled. In the case of NC State, all of its VCL images are equipped with middleware that allows the users to access NC State enterprise storage, storage on their own access computers, as well as any other network-accessible storage for which they have appropriate access permissions.

In production, VCL distinguishes between two types of resources: undifferentiated and differentiated. Undifferentiated resources can be reconfigured and reloaded at will, and end-users can be granted full administrative privileges on the VCL images they have reserved. Differentiated resources are pre-configured and can be made available to the end-user at will or on schedule. VCL differentiated services can be used with certain privileges but generally not modified at the administrative level. Examples of differentiated resources include teaching lab computers that are adopted into VCL when they are not in use (e.g., at night), and other external resources and services that can be connected through the client-side VCL daemon or API. More detailed information about VCL user services, functions, security and concepts can be found in [1, 8, 9].

Currently NC State’s VCL is serving a student and faculty population of more than 30,000. Delivery is focused on augmentation of the student owned computing with applications and platforms that students may otherwise have difficulty installing on their individual laptops or desktops because of licensing, application footprint, or similar. The latest statistics show that the NC State VCL serves over 80,000 reservation requests (mostly of the on-demand or “now” type) per semester, and over 7,000,000 HPC CPU-hours per year. A typical user reservation is 1-2 hours long. There are currently more than 150 production images and another 450 or so other images on the VCL system. Most of the images serve single user requests and HPC cycles, with a smaller number focused on Environment-based (aggregates of images that can be used to form virtual clouds) and Workflow-based services.
At NC State, the demand load on the VCL computational systems is monitored with care. Over the course of a calendar year, non-HPC usage shows recurring peaks and troughs in the demand load. For example, during semester there is a pronounced rise in the user demand level for VCL non-HPC resources. Figure 2 illustrates this periodic demand pattern for the period of September 2004 through February 2009. Inspection of the usage pattern data shows that such demand levels are reached during specific times throughout the year. Not surprisingly major troughs can be identified as corresponding to summer and winter holiday time periods.

Addressing the architectural design considerations of VCL to support these peaks and troughs is a key aspect to building an effective, efficient and economical IaaS. If the only VCL design consideration was to deliver and maintain the necessary hardware capabilities to exclusively service these peak demand loads, it would leave large fractions of the VCL idle for extended portions of the year. On the other hand, having insufficient resources at peak periods will lead to dis-satisfied users because they cannot access and schedule computing resources when they are needed. Having such a large excess (standby) computing capacity to assure availability of cloud resources at all times is not an economically viable path unless “idle” resources can be re-purposed to other alternative uses while they are not needed for VCL desktop student usage.

The VCL implementation has the characteristics and functionalities considered desirable in a cloud computing configuration with the additional capabilities beyond those typically found in the commercial sector. Taking a commercial cloud offering for comparison, functionally VCL has capabilities found in the Amazon Elastic Cloud [11]. It is open source [12], and allows users to construct their own cloud services. For
example, by loading a number of resources (virtual or real) with Hadoop-enabled images [13] one can implement a Google-like map/reduce environment, or by loading and Environment or group composed of Globus-based images one can construct a sub-cloud for grid-based computing, and so on. VCL can also integrate other clouds into an overall VCL resource pool through its API and gateway images.

At NC State, a typical bare-metal blade serves as many as 25 student seats – a 25:1 ratio – a considerably better ratio than traditional physical labs at 5:1 to 10:1. Hypervisors and server-apps can increase utilization by another factor of 2 to 40 depending on the application and user profile. The personnel support costs for this system require about 2 to 3 FTEs in maintenance and help-desk for about 2,000 nodes, with another 3 FTEs in VCL development.

2.2. Measuring the Economic Effectiveness of Infrastructure as a Service (IaaS) in an Educational Environment

For a university based cloud computing system to be economically viable, it requires a scheduling process that carefully shepherds these resources in a way that efficiently and economically matches the demand load over time to the system resources. A typical university environment supports the academic programs that typically see large growth in user demand during assignment times, and perhaps near the end of each academic term. On the other hand, during fall and spring break, during winter holidays, and perhaps in the summer, academic demand can be considerably lower. Universities with a sizeable research presence on the campus however, have research projects and activities that are active year round and show less dependency on the academic calendar. Because research projects that use HPC are chronically short of computational resources, they are an excellent resource utilization backfill, provided that the cloud can dynamically transfer resources between single-“seat” and HPC use modes.

Figure 3 shows the number of non-HPC concurrent VCL reservations for the same period as Figure 2. Figure 4 shows the total and the maximum number of concurrent reservation for November 2008. As Figure 4 clearly indicates, at any given time during the month of November, the maximum number of concurrent blade reservations by day remains steady around a value of 300 to 350. From Figures 3 and 4, we see that in our case concurrent usage is about 20% to 25% of the daily usage. It is important to note that this fraction does depend on the operational profile, so a large number of concurrent class (group) reservations may increase this fraction. Figure 5 shows that the average daily demand (for November 2008) itself varies by the hour of the day, thereby further refining the time window where a larger number of blades need to be operational and available for VCL users.

This average shows the same consistency when measured over longer periods of time. This data clearly suggests that it is not necessary to keep 500 blades active and available for VCL use to provide demand surge protection. This identified spare capacity is the resource pool in VLC that can be re-purposed to other user demands and computational activities on the campus. Having some level of excess capacity or slack in the system is very advantageous for several other reasons. A spare capacity allows for maintenance flexibility and upgrades without disruption to the production systems. It also provides options to support other types of projects that can take advantage of intermittent levels of spare CPU cycles.
High performance computing is an excellent candidate to absorb the spare capacity in the VCL system. In the past year, the on-campus HPC demand has consumed over 7,000,000 CPU hours. Figure 6 shows the plot of the demand load for high performance computing as a function of time by month over the course of the prior twelve month period. Most of the demand is of the batch type. The data shows that throughout the year, the demand for HPC computational resources remains relatively constant. The slight peak during the summer represents additional throughput realized by re-allocating VCL blades that were supporting non-HPC services during the academic year.

The data clearly indicates that by shifting some of the relatively constant but high computing demand HPC load onto spare capacity cycles of the non-HPC VCL resources, it is possible to make more efficient use of the VCL system, minimize the fluctuation in unused non-HPC VCL capacity over time, and provide the HPC computing systems with incremental boosts of computational power.

To understand the economics of load balancing between non-HPC usage and HPC computing within VCL, it is important to look at HPC and the non-HPC operational profile components. Current yearly non-HPC VCL usage is approximately 160,000 reservations and over 300,000 hours. At any one time, up to 500 blades of the VCL cloud are in the non-HPC mode. Current yearly HPC VCL usage is approximately 7,000,000 CPU hours on approximately 500-1,000 blades (most of them are two-processor variants with one, two or four cores each). Support for the desktop and HPC services is interconnected and includes a hardware support person, a system

![Figure 3. VCL concurrent reservations - from September 2004 through February 2009.](image)
administrator, three developers, help desk support, and one person for HPC code optimization.

Based on the current level of annual usage of about 300,000 hours of non-HPC VCL services and approximately 7,000,000 hours for HPC VCL usage, the total cost of ownership (TCO) has been carefully measured for both the non-HPC and HPC VCL usage. This total cost includes hardware refresh, software, licenses, space, power, cooling, and related personnel and support costs to deliver the described VCL IaaS at NC State.

By analyzing the cost data from operations, it shows that a non-HPC VCL session providing a dedicated blade assigned to each user with no virtualization accumulates a total cost/hr that equates to $1.46. By relaxing the restriction of dedicated blades without virtualization enabled, and by allowing a 15% time sharing of the VCL blades for non-HPC sessions the total cost/hr to operate non-HPC VCL services drops to $0.59. Blades that are multi-core and can allow virtualization to be enabled without degradation of service and performance have also been measured. With a virtualization factor of 2 enabled and a 15% time sharing of the blade for non-HPC sessions, the cost/hr drops again to $0.30. If the virtualization is increased to a factor of 4, the cost/hr drops to $0.15. Because the non-HPC VCL usage is still experiencing a superlinear growth, and our new blades (eight cores, 16 GB of memory) can safely host as many as 10-16 virtual machines per blade, we expect as much as a total 10-fold drop in the non-HPC CPU-hour cost as the capacity fills up compared to the dedicated blade assignment without any virtualization enabled.

Further, inspection of the non-HPC and HPC individual modes of service can provide some useful insights. If only non-HPC VCL services were provided today, the
cost/hr would range anywhere from $1.46 to $0.15 based on the percent of time sharing of the blade and the level of virtualization. If only VCL HPC services (using bare-machine loads) were provided today, then the total capital and operational annualized cost would be approximately $1,400,000 to service about 7,000,000 CPU hours. This translates to a cost/hr of $0.20. By utilizing both time sharing and virtualization to include both non-HPC and HPC resources, there is an up-front reduction in the total cost required to individually maintain both of these services separately, and also an increased average utilization of the total available VCL resources that lowers the overall cost per CPU hour.

3. Observations and Insights

Several observations and insights can be gleaned from an analysis of the faculty and student academic and research IT user requirements and how the specific architectural choices in the design of the VCL have generated a more cost effective and better utilized composite system. Probably one of the most important trends that can be inferred from the analysis of the combined non-HPC VCL services and HPC utilization data is in the area of efficient utilization of the computational infrastructure.

The desktop Virtual Computing Laboratory serves an important function, delivering both educational IT support as well as providing small desktop analysis resources for research data. In order to provide these capabilities to users across widely
varying demand loads, it requires that the university make a large capital investment to assure this “on-demand” level of service is available. The user demand over time for the educational aspects or non-HPC VCL resources is governed by the academic calendar of the university. Therefore, when users are able to access these academic cloud computing services with on demand reliability over 96% [8, 9] throughout the academic year, it means that a considerable amount of equipment needs to be in standby, or idle mode, for long periods of time, yielding a low average utilization rate over time and an expensive total cost of ownership for the university.

One of the key VCL design considerations was to integrate the HPC VCL computations into the non-HPC resource delivery, thereby providing an option to markedly decrease the total cost of ownership for both systems. By co-locating a potential complementary computational HPC mode with a higher and more consistent utilization rate over time, and seamlessly integrate the two systems, better utilization is achieved with considerable economic benefits.

The VCL operational statistics over the past several years strongly support this design choice and suggest that by building a coherent integrated campus IT layer for faculty and student academic and research computational needs, it will allow the institution flexibility in servicing both of these university functions. It also allows the educational institution itself to maximize the return on their capital investment in the IT equipment and facilities and decrease the total cost of ownership.

Today many universities have the non-HPC and HPC activities disconnected. The probability for university teaching and learning activities to economically provide a strong IT capability that complements and supports the classroom work is enhanced if
there is also a large strong research component on the campus utilizing a common research computing infrastructure. Our paper indicates that the incremental cost to provide both efficient and economical academic and research computing services with minimal underutilization of equipment is enhanced by integrating the university IT teaching and learning aspects into the same capital equipment infrastructure that serves the HPC computing cloud needs. In particular, at NC State, using the VCL blades for both HPC and VCL desktop work provides economical services with minimal underutilization of equipment. A VCL implementation is a major step in achieving a coherent CyberInfrastructure from the desktop through the campus layer and beyond.

If there is no large research computing user base, it may still be possible to achieve an efficient utilization of resources applied to desktop virtualization. However, this requires a much larger and diversified base of users to effectively utilize the large solid computational base in the business model that will allow a fraction of the users at any particular time to have the on-demand desktop VCL capabilities. An important aspect to consider in any integration of such capabilities is to make sure that the additional services not all have the same user demand cycle over time. It may be possible to construct a diversified user base from a combination of K-12 users, community colleges, and university teaching and learning students that each operate on a slightly different academic calendar. Measurements from implementation within the North Carolina Community College System (NCCCS) demonstrated it may be possible to realize savings of 50% in the infrastructure budget [6].

A much longer and more detailed analysis of the economic model and in-depth development of the business case for VCL cloud computing in a higher institution education environment utilizing the experience and operational data from the NC State Virtual Computing Cluster is in preparation at this time. This analysis will also include a comparison of a VCL educational implementation to the current choices among commercial cloud computing providers.

The data presented here indicates that there is a strong economic case for configuring an educational institution’s computational resources using an Infrastructure as a Service paradigm and then layering PaaS, AaaS and SaaS services below as needed. The increased efficiency and cost reductions of such an implementation should be given consideration by educational institutions seeking cost savings to provide better educational academic computing while also increasing the research computing service capacity for the campus.

Acknowledgments

This work has been supported in part by IBM, Intel, NetApp, Cisco, NC State, UNC General Administration and State of North Carolina. We would like to thank all our colleagues on the VCL project, and especially Andy Kurth, Aaron Peeler, Henry Schaffer, Sarah Stein, and Josh Thompson, for their input and support.

References


[10] Sam Averitt, E-Research In the Clouds, Educause Net@Edu, 2009 http://connect.educuse.net/Library/Abstract/EResearchintheClouds/48135


Facing Services in Computational Clouds

Thijs METSCH a,1, Luis M. VAQUERO b, Luis RODERO-MERINO b, Maik LINDNER c and Philippe MASSONET d

a Sun Microsystems, Germany
b Telefónica Investigación y Desarrollo, Spain
c SAP Research, United Kingdom
d CETIC, Belgium

Abstract. When describing some parts of a big system they all seem so different. But combining them all together leads to the big picture. The same is true for today’s technologies. When looking at all the different facets of Grid and Cloud computing no concrete picture might evolve. But trying to address all kind of issues might lead to a system which is able to manage service on demand. This paper tries to provide the big picture, of what today is called Cloud computing. It focuses on the faces, business models and handling of services in Clouds. And therefore tries to give the big picture of Cloud computing.

Keywords. Cloud computing, Utility computing, Grid computing

Introduction

What today is called Cloud computing comes a long way down the road. It had other names before and many technologies are involved in it. Virtualization, utility computing, and grid technologies are among the most representative. The separation among these technologies is diffuse, which hardens to find an unique, comprehensive definition of what Cloud computing is, although some attempts to obtain a consensual definition have already started [41]. In a broad sense “Clouds offer the ability to deploy a series of functionalities in the form of services that use a wide variety of heterogeneous resources in a pay per use manner” [27].

Cloud systems can be classified according to the resources they offer ‘as a Service’ (XaaS): Infrastructure as a Service (IaaS) that allows to allocate virtual machines and storage capacity; Platform as a Service (PaaS) where users are provided with remote software platforms to run their systems; and Software as a Service (SaaS) where applications that traditionally have run in the desktop are now moved to the Internet and accessed through web interfaces.

Arguably, IaaS systems are the ones that have achieved the greatest impact on the ITC field so far. There is a wide range of commercial solutions based on such paradigm, such as Amazon EC2 and GoGrid, that allow to allocate hardware infrastructure in their external Clouds. Other approaches like Rackspace, ServePath and DataSynapse’s Feder-
ator allow the combination of dedicated servers with infrastructure deployed on external Clouds. It is worth to note how IaaS systems are evolving to provide more advances services, beyond pure infrastructure. For example, Amazon recently announced automatic scaling capabilities similar to those offered by Rightscale and similar platforms like Scalr or WeoCeo.

Also, scientific Clouds like OpenNebula [3], Nimbus or Eucalyptus [2] allow to use locally owned data centers with external Clouds for automatic provision of IT infrastructure.

Service denotes that applications offer some added value to the customers. This paper tries to categorize the different kinds of services that can be deployed in a Cloud. Services need to be virtualized and described for deployment. Different kind of services bring different deployment models, abstraction levels and virtualization methods [20].

The Cloud usage patterns are useful tools for tracing the roadmap from simple Service Oriented Computing to the provision of services on Cloud environments. The transition from monolithic, centralized applications to distributed services offered in a Cloud environment has not been, however, straightforward. Two technological advances can be remarked. First, the need for huge scalability has been both a beneficiary and a trigger of an increase in the scaling capabilities of the underlying supporting infrastructures. In addition, more recent trends like Web 2.0 have brought new interfaces that ease the management of platforms and resources.

However, these interfaces offer abstractions that are still too close to the infrastructure, while the usage patterns of services in the Cloud show integration need to build complete and usable applications for the end user. New interfaces and abstractions, closer to the companies business models, are required, which in turn implies changes in the service delivery mechanisms. Also, new security concerns for these new delivery approaches have arisen that were previously unattended or disregarded. A relevant ongoing task related with the creation of these new abstractions is being developed within the Resources and Services Virtualization without Barriers (RESERVOIR) project that further illustrates these brand new challenges [4], which we also present.

So overall the paper tries to address how services can be addressed in Clouds looking from multiple points-of-views including, but not limited to, evolving business models, type of services and emerging patterns for Clouds. The rich variety in types and deployment of services is referred to in the title of this paper by the meaning of ‘facing’ services in Clouds.

The remainder of this paper is organized as follows. First a general overview of the different incarnations of a service in Cloud environments is presented (see Section 1). After presenting this variety of Cloud services we also present some data regarding the paramount usage patterns found in current Clouds (Section 2). Then, section 3 presents the exploitation mechanisms employed by some well-established players in the Cloud markets. This will help to understand their business models. These models are strongly influenced by security mechanisms that should be enabled to promote a wide Cloud adoption in the IT community (see Section 4). Section 5 comments the basic features that Cloud environments should implement. Finally, section 7 further emphasizes the advantages of Cloud services and presents some challenges ahead that will need to be solved to obtain a full implementation of Cloud services in the enterprise.

---

2The research leading to these results is partially supported by the European Community’s Seventh Framework Programme ([FP7/2001-2013]) under grant agreement # 215605.
1. The different faces of a service

Cloud computing is triggering a new way for provisioning computing services, i.e., everything can be viewed as a service in a Cloud. This new paradigm shifts the location of the services/resources to the network, which can be seen as a huge pool of virtual resources which are allocated and bought in on a pay-per-use manner, reducing the costs associated with the acquisition and management of the required hardware and software [23].

Depending on the type of provided capability, there are several types of Cloud services [27], the most representative are:

1.1. Infrastructure as a Service

Plainly speaking, Infrastructure as a Service (IaaS) products deliver a computing hardware infrastructure over the Internet. IaaS providers offer a virtually infinite set of computing resources. Virtualization technologies enable them to split, assign and dynamically resize these resources to build custom infrastructures, just as demanded by customers.

Apparently, this service does not differ much from the classical hosting service. What makes the Cloud a novelty is the self-management capabilities it offers: The possibility of an almost immediate resizing of the assigned resources, and the application of the pay-per-use revenue model.

Some of the most representative examples of such services are (Amazon EC2, Joyent, SUN Microsystems’s Network.com, GoGrid, HP Flexible Computing Services, 3Tera, etc. [27]).

1.2. Platform as a Service

In addition to this “very advanced hosting”, the services in the Cloud can offer an additional abstraction level: rather than supplying a virtualized hardware infrastructure, they provide the software platform where customer services run on (e.g. a LAMP stack). This is denoted as Platform as a Service (PaaS). An important feature of PaaS systems is that the sizing of the hardware resources demanded by the execution of the users services is made by the PaaS provider in a transparent manner.

Probably, the best-known example is the Google Apps Engine [1] (although many other competitors also offer their development environments as Cloud services, like, for instance, Coghead, LongJump, Etelos, salesforce.com, etc.) [27].

1.3. Software as a Service

IaaS and PaaS systems have in common their aim to be a platform for their users. Software as a Service (SaaS), in contrast, groups together those Cloud systems in order to create a final aggregated service themselves. These services are software products that can be in the interest of a wide variety of users. In other words, these products offer a complete turnkey application (even very complicated ones) via the Internet.

There are several savings in this approach: the software maintenance costs are nullified for the end user and the software vendors avoid performing multiple installations of the same code in their clients’ facilities. This is an alternative to locally run applications.
An example of this is the online alternatives of typical office applications such as word processors. Oracle’s SaaS, SalesForce Automation, NetSuite, Google Apps, etc. are the most representative examples of this type of Cloud services [27].

1.4. X as a Service

Many other resources can also be offered as Cloud services, such as Storage as a Service, Messaging as a Service, Ethernet as a Service (which allows customers to connect remote LAN segments into a virtual Ethernet LAN [36]), etc., although they are usually just particular types of one of the three groups of services mentioned above (IaaS, PaaS, and SaaS). For example, Messaging as a Service can be regarded as a kind of PaaS service. The term XaaS has been coined to refer to any kind of resource in the Cloud.

1.5. Virtualization and abstraction

Different deployment models require different ways of abstracting services. Cloud services in a IaaS manner are rather easy to virtualize. Being mostly based upon common virtualization technologies like KVM, VmWare or Xen. The user only supplies the virtual image in which the services and the operating environment is deployed.

In a PaaS approach only the services are deployed by the user. The virtualization technology and the operating environment is provided by the Cloud provider. The complete environment, service deployment and virtualization is hidden when taking the SaaS model in account.

These different levels of virtualization require different levels of security and abstraction. Working in an environment in which Cloud providers deploy nothing but virtual images (IaaS), security considerations are very important. Less important are restrictions to the service, since the complete environment is provided by the customer.

When the service provider does not bring the full image, but only development services are deployed (PaaS), security handling becomes easier. But the diversity of the environments in which the service can be deployed is restricted by the infrastructure offered underneath. Finally, if only usable services are offered (SaaS), both the security considerations, as well as the diversity are both in the hand of the Cloud provider and therefore easy to manage.

2. Emerging patterns for using Clouds

In sight of all these paradigms, it is not easy to derive out some preferential “usage patterns” in Cloud services. This is due to the broad range of services available, each one targeted to different users needs.

However IaaS systems where the ones to trigger the interest in the new Cloud paradigm, and are arguably closest to the Cloud concept in the ICT professionals mindset. IaaS systems are used by Small and Medium Enterprises and start-ups that cannot afford the acquisition and management of their infrastructure. More over, IaaS solutions are used to supply companies with additional resources for limited periods of time, for example to address peaks on services demand, or for testing purposes. Thus, companies avoid getting extra hardware resources that could be underused in the future.
Regarding SaaS, users have focused on the deployment of Web 2.0 applications offered in a SaaS manner. Some well-known sites offer the user the chance to develop simple applications (a la PaaS) and offer them in a SaaS-like manner later on. This usage pattern could also be called extension facilities.

PaaS is an optimal environment for users seeking testing and development capabilities, these are two new emerging use patterns which are gaining popularity. Probably, gaming will be one of the most remarkable usage patterns for Cloud technologies, due to their inherent scalability, endowing them with virtually unlimited graphical power and players.

Also the rise of netbooks in the computer hardware industry triggered the development of Clouds. These slim devices depend on services being deployed in remote Cloud sites since their own capacity is limited. Behind this stand the idea of getting access to everything, from anywhere, at any time.

Commonly not only the application but also the deployment of service in a Cloud follow certain patterns. Widely know Cloud service providers like Amazon are described as public Clouds. They are freely accessible while private Clouds are deployments of the Cloud infrastructure within a company. Mostly they share the same interface and some projects like OpenNebula or Eucalyptus [2] try to create hybrid Clouds. In a hybrid Cloud public cloud resources are added to the private Cloud to serve the demand for resources.

All of the previously described usage patterns enabled by the aforementioned Cloud models result in the proliferation of new services. Thus, a huge variety of newly generated services are being deployed over the Cloud. These services need to be integrated not only with the Cloud, but also among themselves to build completely usable applications that satisfy customized user needs. One of the most outstanding necessities of users is having an appropriate supply chain. The adoption of new ways of delivering services is directly translated into changes in the companies supply chains. Business models are appropriate tools to find out whether the costs incurred with the modification of the supply chain are compensated by the revenue produced by the exploitation of the newly generated services.

3. Evolving dynamic business models and market framework for Cloud services

3.1. Single enterprise view

From an economic perspective, every Cloud computing service is based on a certain business model. These models are, in general, used to describe the value-creating logic of organizations within a certain market, e.g. The means by which a company or network of companies aims to make money and create consumer value with Cloud offers. Thus, they should be the basis for every company’s business. As a matter of fact, in today’s dynamic, highly flexible and mostly technology-driven ICT market, the underlying business models are often “just” created to fulfill requirements from investors and banks. Nevertheless, a valid business model bears advantages for every company, especially on rapidly growing and emerging markets like the Cloud computing service market.

From a scientific perspective business models serve various purposes [30,31]: understanding the elements and their relationships in a specific business domain, communicating and sharing this understanding to the outside world, using them as a founda-
tion for change, measuring the performance of an organization, simulating and learning about the business, experimenting with and assessing new business models, changing and improving the current way of doing business.

Since its inception, the field of Business model research has developed from defining the concept, to exploring business model components, developing taxonomies of typical business models, and developing descriptive models [31]. So far, the field has not established a single common definition of a business model[7]. The definition of Chesbrough and Roosenbloom who see a business model as a blueprint for the way a business creates and captures value from new services or products can be seen as most comprehensive and suitable for the area of ICT [16]. Also external factors such as socio-economic trends, technological developments, and political and legal changes play an important role in understanding how business models are used developed and used in practice.

Arriving at methods of classifying business models one has to mention Bambury’s approach[11], which distinguishes traditional businesses from alternate online models and second Timmers’s approach[38], which focuses on the business processes whereby the degree of integration and innovation are the prime criteria. Other approaches like the ones from Rayport [33], Jeon[25] or Mahadevan[29] examine similar criteria with slightly different aspects.[35] With regards to the value analysis of business models Amit and Zott[9] proposed the following factors of value creation for (e-)business: efficiency, complementarity, lock-in and novelty.[35] In order to come up with a fitting analysis and evaluation of an existing and novel business model it is most definitely required to take various classifications and success factors into account.

A well-known, and probably best fitting framework for evaluating business models for Cloud computing services is provided by Afuah and Tucci.[6] The business component of this framework consists of the following seven pillars:

customer value The selling point to the customer is always the value, which can be obtained by implementing a certain ICT solution.

business area Every company should be clear about the business area they are acting in.

A clear structure of the offered services helps to clarify here.

price A fixed price schema is the basis for every market transaction.

source of revenue Highly linked with the price and the business area the sources of revenue need to be clearly defined to ensure the sustainability of the model.

linked activities Not only the core business (Cloud services), but linked activities should be taken into consideration for a business model. This might lead to extended business at a later point in time.

capability of execution Planned activity on the market should be backed-up with a solid plan on how to actually execute the business. This includes an assessment of key indicators like market size or expected demand.

continuity Every business should strive for continuity, as investments and other obligations are based on a longterm perspective.

This framework especially fits the Cloud computing services needs, as it highlights the relevant areas of an emergent market. So each company acting on the Cloud service market should be able to describe their business model based on these seven main pillars. Once a company has established the business model and started going into business, this business model needs to be adapted on a regular basis according to external and internal changes.
With a few exceptions [10,28,40], most literature has taken a static perspective on business models. They are used to describe the value-creating logic of organizations at a certain moment in time. Hereby the implicit assumption that business models remain steady over time, and that choices are rarely corrected. However, in reality business models do not persist forever. Organizations often have to review their business model in order to keep in line with fast changing environments.[5] This is particularly the case in industries experiencing rapid technological advances like the ICT sector. As a result, de Reuver et al. strongly request that business models have to keep up with external changes during all phases from development to exploitation.[18] Following this argumentation the development and practical implementation of dynamic business models for Cloud computing services have to be fostered. This concept is of even more theoretical and practical relevance hence Cloud computing as a rapidly growing, emerging technology is definitely underlying the rules of hype cycles as introduced by Gartner.[22] As such, the whole business including market players, products and business implementation needs a dynamic perspective. In order to achieve this a market specific phase model needs to be introduced.

Generally, phasing models help to understand how innovation and change affect the evolution of markets and related business models.[6] Phasing models have appeared in technical service development, entrepreneurial and business planning, innovation adoption, and marketing. As argued by Kijl et al.[26], these models widely imply three main phases: technology/R&D, implementation/roll-out, and market (the latter consists of the sub-phases market offering, maturity, and decline). The phases compose a circle, each triggering the following phase.

Technology is the most important driver for the urge of continuous adoption of business models in the ICT sector. The emergence of new mobile, wireless, and data networks enable increased reach of businesses while at the same time middleware and multime-
dia applications offer new opportunities for enriched, customized, and secure communication. However, it can be assumed that market developments and regulation can also push opportunities for the development of new products and services. Changes in market chances or regulations from Government side enable new product and/or service definitions and therefore new underlying business models. As the current hype cycle phase according to market development is of high relevance for the business model, it builds the fourth influence factor. According to the hype cycle this proposed phase model [18] needs to be applied to all different phases of the cycle.[12]

This phase model represents the dynamic influence factors on single enterprises’ business models. In order to create a full market view it is needed to introduce a market model. This then will partially support the business model of each individual company. As visualized in Figure 1 it shows which external drivers are expected to play a major role throughout the dynamic adoption of business models.

3.2. Market view

Porter offers a framework for industry analysis and business strategy development.[32] Within this framework the actors, products and business models are described and their interaction on the market structured. This leads to a comprehensive picture of the business side of Cloud computing services (comp. Figure 2).

The Market Model of Porter (as part of the whole industry framework) consists of five influencing factors/views (forces) on the market: Market, Suppliers, Buyers/Consumers, New Market entrants and Technology development. In the traditional economic model, competition among rival firms drives profits to zero, thus forcing firms to strive for a competitive advantage over their rivals. The intensity of rivalry on the market is influenced by industry-specific characteristics[32]:

![Figure 2. Business Framework for dynamic Business Models based on Porter.[32]](image)
The amount of companies dealing with Cloud and Virtualization technology is quite high at the moment. Thus, the rivalry is quite high. On the other hand the products and offers are quite varied, so many niche products tend to establish.

The Cloud computing service market is presently booming and will keep growing during the next few years. Therefore, the fight for customers and struggle for market share will begin once the market becomes saturated and companies start offering comparable products.

The initial costs for huge Data Centers are enormous. By building up federations of computing and storage utilities smaller companies can try to make use of the economy of scale effect as well.

In order to avoid vendor lock-in Cloud service users tend to choose services, which are based on standards and market-wide accepted interfaces. In the case of Cloud offerings, it is not just about standards for the service-user interface, but also about the service-service interface. Most current Cloud offerings are only paying attention to standards related with the interaction with the end user. However, standards for Clouds interoperability are still to be developed[19].

The combination of growing markets and the potential for high profits tends to lead to a shakeout, which accompanies intense competition, price wars, and company failures. Monitoring the Cloud market and observing current trends will show when the Shakeout will take place and which firms will have the most accepted and economic offer.[34]

Together with the single enterprise view on emerging ICT markets this market view allows to create a full picture of the Cloud service Market.

Some authors claim that Cloud computing still has huge issues.[24,39] First, that business and IT managers of critical applications are rightly concerned that a shared virtualized environment means they could lose operational control over their application performance and location.[13] The reason: virtualized environments are often shared by design, so when conflicting computing demands arise, whole services might be migrated across subnet- or even country-boundaries. The implementation of Service Level Agreements (SLA) management techniques at several levels is a key technique for building a trust worthy Cloud computing market.[14] Nevertheless, the security of Cloud computing services is a technical as well as an economic key issue for Cloud computing services. Thus, Cloud operators need to find security models that mitigate the risk for Cloud service users.[17]

4. Security considerations while using services in Clouds

Many of today’s companies use modern technologies to ensure that valuable data stays in-house. Although with the upcoming ideas which are described in the previously section, Cloud computing becomes interesting for companies and firms. Therefore security considerations play an important part. While the security area has a rich variety of facets it is not the intention of the authors to go into deep detail. The following section will deal with them and address technical as well as Political issues of Cloud computing in a broad manner.

Large-scale, cross border, virtualized service infrastructures present a promising approach to cope with the ever increasing requirements of modern scientific and business
applications. Unfortunately, these infrastructures also provide new opportunities for malicious users to find vulnerabilities to attack and exploit the systems. To deal with these threats adequate security safeguards for virtualized infrastructures need to be provided for the different parties such as infrastructure providers, service providers and end-users.

Different fields of security have to be taken into account.

**Trust** Intellectual property of companies and persons need to be protected. This is true for licenses and data as well as processes which are realized, encapsulated and inherited in services and data storages. The management and guarantee by which the trust relationships are realized are managed in different ways by providers.

**Deployment** The way the service or the data is deployed to the resources of the provider can lead to trust and security issues.

**Handling** Meta-information, which is provided next to the service, can be interpreted in different ways. The interpretation method is realized by the Cloud provider and, therefore, is questionable.

**Isolation** At the same time how the services are isolated from each other forms a major security concern. Different virtualization methods lead to different levels of isolation.

**Data movement** When service are migrated between sites, Clouds and resources the question of delegation comes up. Also the transfer methods need to be evaluated.

In addition to these, internal and external security threats exist. External threats are linked to communication across Cloud sites. Those threats related to communications are well know threats such as threats linked to men-in-the-Middle, TCP hijacking (spoofing), migration and security policies, or identity theft. These threats aim to gain unauthorized access to the Cloud and to impersonate entities. These techniques allow the attackers to eavesdrop as well as to modify or copy data.

The external Cloud interfaces are also exposed to threats. The external interfaces can be subject to the following attacks: denial of service (DoS or Distributed DoS), flooding, buffer overflow, and p2p-attacks. These attacks are aimed at trying to use the limited resources or force a system crash, leading to the inability to perform ordinary functions.

Clouds must also protect themselves from internal threats. Since Clouds provide infrastructure services to the public, many external users are granted access to the Cloud infrastructure. So even though users are authenticated and authorized to access the Cloud infrastructure, a Cloud computing infrastructure must also be protected against internal threats.

5. **Overview of a Cloud computing framework**

Different faces of services in a Cloud lead to high requirements with regard to the administrative, business and technical environment. A Cloud computing framework which wants to manages and host all kind of services like described in section 1 has to fulfill these requirements and deal with them in an optimal manner.

A basic environment should have the following characteristics. A Service Definition Language should enable the management of service across different sites. Beside being a language to support deployments it also enables to manage the life cycle of services.
Algorithms in the RESERVOIR environment will ensure that the allocation of resources conforms to defined Service Level Agreements.

Security mechanisms for safe deployment and relocation of services will be implemented. This also means that end-to-end security has to be supported when accessing services.

The previously described business models (See section 3) also demonstrate the need for accounting. Billing mechanisms will be available to charge for resources used.

With services becoming used more and more in today’s communities, a need for open environments arises. Deployment of services in modern Cloud sites is covered by virtualization technology. The on top added value is the management of the deployed services. Services can be moved and migrated, suspended, resumed, started and stopped. Each of these processes can be triggered by Service Level Agreements. The environment should take care that the deployment of the services, is in compliance with the installed SLAs. This can mean that similar instances of services have to be grouped together on one physical resource for performance optimization.

Other possible SLA policies might include:

- Keep instances of services, which belong to one organization, on the same physical resource
- Turn off physical resources which are not needed, to reduce energy consumption.
- Try to move all instances which belong to the same organization to the organization’s resource.
- Keep the number of services running on external resources low. This can reduce costs since most services are running locally on the own resources.
- Move services to resources which are geographically near to the end-user.

All of these policies and operations can lead to performance increases, cost reductions and power efficient usage of resources. And therefore, create an elastic environment for services in Clouds.

Furthermore in extension to the technical requirements, the environment has to incorporate various business models and support their execution. Therefore the provision of adequate mechanisms to monitor, track and steer the faultless execution of operative business needs to be in place. Among partners it is important to create visibility and clarity about business goals, their linked activities, their pricing models, and especially their capability of execution. This is crucial for a close collaboration in an, even “business-elastic” environment.

The RESERVOIR project tries to create an environment in which services can easily be deployed and managed. This environment provides an abstraction layer from specific resources, platforms and geographical locations.

Beside the major idea of creating abstractions, the project tries to create an environment in which services can be migrated. This includes the migration of virtual machines as well as for example Java services. Migration should be possible across network and storage boundaries. In this case, both a “live” and a “suspend and resume” method of migration are supported.

RESERVOIR encapsulates all of these ideas of an environment, in which different kind of services can be managed. It tries to create a scalable flexible environment for hosting services. This environment should be build with modern technologies and open standards. Such an environment with suitable utilities can improve future data centers.
5.1. The RESERVOIR environment

Two different user groups are needed to manage all kind of services in a Cloud environment. First of all the service end-users need to access their services. Secondly the service administrators need to configure the system and define SLAs. So the overall system for a Cloud site may resemble to that one shown in figure 3.

The service administrator and the service end-user use a different interface to access the overall environment. The end-user directly talks to the service. The service itself is abstracted from the resources. To ensure this kind of virtualization a management environment is also needed.

Deployment can be done by the service administrator or a service provider which may or may not be the same person. So the service management interface can also be divided into two subsections. One for deployment and one for management of the Cloud site. For simplicity this is not considered in this paper.

This being the basic operations in a Cloud based environment still some specific ideas of RESERVOIR are needed. Some special abstractions are made and terminology are used. These are explained in the following paragraphs [15].

While services can have so many different faces the first abstraction made in the RESERVOIR project is the encapsulation of services. One service or a group of services can run inside a Virtual Execution Environment (VEE).

Several VEEs can then run on a Virtual Execution Environment Host (VEEH); meaning one physical resource. Each host has a virtualization technology installed in which the VEEs are deployed and hosted. While one VEE can consist of more than one service, VEEs can be collocated on one or spread across several VEEHs. A small component for management and monitoring is also available on all VEEHs.

The overall management of all VEEHs is realized in the Virtual Execution Environment Management (VEEM) system. It is in charge of the deployment of VEEs on top of VEEHs. It can bootstrap and unload VEEHs. VEEs can be moved around and placed according to the VEEM setup. The VEEM is totally transparent to the end-user. He does not interface with this system at all.

To complete the overall system, a Service Manager is needed. It is responsible for the instantiation of service applications. It, therefore, requests the VEEM to create and manage VEEs. Beside this the Service Manager also tracks and manages the SLAs. It ensures that all SLA policies are satisfied at all times. To do so, it monitors the current state of the overall system and executes the elastic rules.
This overall architecture is a basic description of how an Cloud environment can look like. However a basic mapping of existing technologies and research projects onto this high level architecture can be done. Solutions like Rightscale or Scalr can be seen as Service Managers. OpenNebula and the Eucalyptus project would fit in as VEEMS. Whereas on the lower layer solutions from Amazon and GoGrid serve their purpose. Within the RESERVOIR project OpenNebula is used as a VEEM where as Amazon’s EC2 Service demonstrates the usage of hybrid clouds. The Service Manager is developed within the project.

Figure 4 shows the overall architecture of RESERVOIR.

Between the different systems, several interfaces are used. The service provider defines a service and a manifest which he can deploy on a RESERVOIR site using the Service Manifest Interface (SMI). The service manager communicates and spins off requests to the VEEM using the VEE Manager Interface (VMI). The VEE host Interface is finally used to control the life cycle of service of VEEs on the VEEH.

The high diversity of the term services (See section 1) brings in the idea to support different hypervisors\(^3\). Within the RESERVOIR project container for Xen, KVM and Java based services are supported.

Repositories in which data and the virtual machine images are deployed are also installed in this system. The abstraction from the network resources is performed by the virtualization software according to the rules and boundaries given by the VEEM.

A complete monitoring and accounting system ensures that the business models can be realized (See also section 3). These features are also needed to ensure that all SLA policies are met. The service manager relies on the information of these parts. All the components here have therefore technologies to support a monitoring and accounting model.

Some of the issues described in section 4 are addressed by RESERVOIR. One of the characteristics is that this environment aims to federate heterogeneous physical in-

\(^3\)A hypervisor is defined as an abstraction layer between system, to allow the deployment of several environments on top of one resource.
frastructures. Security threats must thus be considered for federations of collaborating infrastructures.

Security threats within RESERVOIR sites can be classified into external threats and internal threats. External threats deal with interactions between service providers and primary Reservoir sites. Internal threats deal with threats to interactions between the components within a RESERVOIR site.

This overall environment tries to demonstrate that it is possible to create an environment in which different types of services can be deployed and managed by the use of abstractions.

Different abstraction levels and different virtualization technologies of services lead to a complex system with many interface. Still a RESERVOIR based environment could be used for IaaS, PaaS and SaaS based models thanks to the abstractions made.

The emerging business models for Cloud Service offerings will be incorporated into the RESERVOIR framework. The idea of a technical environment, that allows various business partners to collaborate and share an administrative, business and technical platform is the basis for future cloud visions. And so, RESERVOIR is aiming for providing such a framework. But nevertheless the implementation is still ongoing and various hurdles need to be passed.

6. Related work

The development of Cloud environments and related business models is focused in many scientific projects. Grid and Cloud computing models and environments share some common methodologies. Ian Foster et al. try to do a comparison of those in their paper [21]. Actual projects like SLA@SOI [37] try to concentrate on the SLA management in Clouds. The SLA@SOI project uses similar approaches and ideas which are used in the RESERVOIR project which was described in section 5.1. The Eucalyptus project [2] is an actual example where a research project used business models and an Cloud environment to evolve into Eucalyptus Systems. Now Eucalyptus Systems is an commercial provider for using private and public Clouds in conjunction. Work done similar to the work described in this paper has been done by Jörn Altmann et al. by describing the taxonomy of Grid Business Models [8].

7. Conclusions and challenges ahead

Offering products as Cloud services presents inherent advantages. Companies, which previously had to buy and maintain their own hardware and software, hire highly specialized staff to look after the system, eliminate the need of these expensive and time consuming tasks. Moreover, Cloud providers often offer redundant systems, which guarantee the service availability. This could not easily be done by small and mid-sized enterprises alone. Furthermore, the fact that services are offered via the Internet allows for a simpler integration of services belonging to different vendors, thus avoiding service-vendor lock-ins. Companies need to keep some excess capacity for their offered services to deal with unexpected peaks in demand, which is very inefficient in economic terms and leads to underuse of resources, idle resources and wasted money.
Although the ability to control costs and provision services of a very heterogeneous nature in a custom easy manner is very appealing, there are many “Clouds” on the horizon regarding the technology’s maturity. The lack of full control over the provisioned services is still feared by IT departments. Data and application security are also significant concerns derived either from the client requirements or from national regulation (the EU does not allow some personal data to dwell outside the EU borders). These are just non-functional reluctance, but we can also identify some operative concerns. Performance reservations can prevent companies from using transactional and other data-intensive applications in the Cloud. While a client can save a lot on equipment or software, they can also incur higher network related costs from their service providers.

The many faces that services can have in a Cloud leads to many problems and complex Cloud hosting environments. In addition, scalable computing, demands on management and security add a surplus to the complexity. Still an environment like RESERVOIR can address all these requirements and create an environment for managing services in Clouds.

References

Aneka: a Software Platform for .NET based Cloud Computing

Christian VECCHIOLAa, Xingchen CHUa,b, and Rajkumar BUYYAa,b,1

a Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
b Manjrasoft Pty Ltd, Melbourne, Australia

Abstract. Aneka is a platform for deploying Clouds developing applications on top of it. It provides a runtime environment and a set of APIs that allow developers to build .NET applications that leverage their computation on either public or private clouds. One of the key features of Aneka is the ability of supporting multiple programming models that are ways of expressing the execution logic of applications by using specific abstractions. This is accomplished by creating a customizable and extensible service oriented runtime environment represented by a collection of software containers connected together. By leveraging on these architecture advanced services including resource reservation, persistence, storage management, security, and performance monitoring have been implemented. On top of this infrastructure different programming models can be plugged to provide support for different scenarios as demonstrated by the engineering, life science, and industry applications.


Introduction

With the advancement of the modern human society, basic and essential services are delivered almost to everyone in a completely transparent manner. Utility services such as water, gas, and electricity have become fundamental for carrying out our daily life and are exploited on a pay per use basis. The existing infrastructures allow delivering such services almost anywhere and anytime so that we can simply switch on the light, open the tap, and use the stove. The usage of these utilities is then charged, according to different policies, to the end user. Recently, the same idea of utility has been applied to computing and a consistent shift towards this approach has been done with the spread of Cloud Computing.

Cloud Computing [1] is a recent technology trend whose aim is to deliver on demand IT resources on a pay per use basis. Previous trends were limited to a specific class of users, or focused on making available on demand a specific IT resource, mostly computing. Cloud Computing aims to be global and to provide such services to the masses, ranging from the end user that hosts its personal documents on the Internet, to enterprises outsourcing their entire IT infrastructure to external data centers. Never

1 Corresponding Author: Rajkumar Buyya, e-mail: raj@csse.unimelb.edu.au
before an approach to make IT a real utility has been so global and complete: not only computing and storage resources are delivered on demand but the entire stack of computing can be leveraged on the Cloud.

Figure 1. Cloud Computing architecture.

Figure 1 provides an overall view of the scenario envisioned by Cloud Computing. It encompasses so many aspects of computing that very hardly a single solution is able to provide everything that is needed. More likely, specific solutions can address the user needs and be successful in delivering IT resources as a real utility. Figure 1 also identifies the three pillars on top of which Cloud Computing solutions are delivered to end users. These are: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure/Hardware as a Service (IaaS/HaaS). These new concepts are also useful to classify the available options for leveraging on the Cloud the IT needs of everyone. Examples of Software as a Service are Salesforce.com\(^2\) and Clarizen.com\(^3\), which respectively provide online CRM and project management services. PaaS solutions, such as Google AppEngine\(^4\), Microsoft Azure\(^5\), and Manjrasoft Aneka provide users with a development platform for creating distributed applications that can automatically scale on demand. Hardware and Infrastructure as a Service solutions provide users with physical or virtual resources that are fitting the requirements of the user applications in terms of CPU, memory, operating system, and storage. These and any others QoS parameters are established through a Service Level Agreement (SLA).

\(^2\) http://www.salesforce.com
\(^3\) http://www.clarenz.com
\(^4\) http://code.google.com/appengine/docs/whatisgoogleappengine.html
\(^5\) http://www.microsoft.com/azure/
between the customer and the provider. Examples of this approach are Amazon EC2\(^6\) and S3\(^7\), and Mosso\(^8\).

It is very unlikely that a single solution provides the complete stack of software, platform, infrastructure and hardware as a service. More commonly, specific solutions provide services at one (or more) of these layers in order to exploit as many as possible the opportunities offered by Cloud Computing. Within this perspective, Aneka provides a platform for developing distributed applications that can easily scale and take advantage of Cloud based infrastructures. Aneka is software framework based on the .NET technology initially developed within the Gridbus project [2] and then commercialized by Manjrasoft\(^9\). It simplifies the development of distributed applications by providing: a collection of different ways for expressing the logic of distributed applications, a solid infrastructure that takes care of the distributed execution of applications, and a set of advanced features such as the ability to reserve and price computation nodes and to integrate with existing cloud infrastructures such as Amazon EC2.

This chapter provides an overview of Aneka as a framework for developing distributed applications and we will underline those features that make Aneka a Platform as a Service solution in the Cloud Computing model. The remainder of this chapter is organized as follows: Section 1 provides a brief introduction to the Cloud Computing architecture and features a comparison between some available commercial options. Section 2 gives an overview of Aneka by describing its service oriented architecture and the fundamental components of the system such as the Container and the core services. Section 3 presents application development with Aneka. In particular, the different Programming Models supported by the framework and the Software Development Kit are addressed. Section 4 provides an overview of the tools available within Aneka to manage the system, deploy applications, and monitor their execution. Section 5 describes some case studies where Aneka has been used to address the needs of scalability for different classes of applications. Conclusions and a discussion about the future development directions follow in Section 6.

1. Cloud Computing Reference Model and Technologies

In order to introduce a reference model for Cloud Computing, it is important to provide some insights on the definition of the term Cloud. There is no univocally accepted definition of the term. Fox et al. [3] notice that “Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and system software in the datacenters that provide those services”. They then identify the Cloud with both the datacenter hardware and the software. A more structured definition is given by Buyya et al. [4] who define a Cloud as a “type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreement”. As it can be noticed, there is an agreement on the fact that Cloud Computing refers to the practice of delivering software and

\(^{6}\) http://aws.amazon.com/ec2/
\(^{7}\) http://aws.amazon.com/s3/
\(^{8}\) http://www.mosso.com/
\(^{9}\) http://www.manjrasoft.com/
infrastructure as a service, eventually on a pay per use basis. In the following, we will illustrate how this is accomplished by defining a reference model for Cloud Computing.

Figure 2. Cloud Computing layered architecture.

Figure 2 gives a layered view of the Cloud Computing stack. It is possible to distinguish four different layers that progressively shift the point of view from the system to the end user. The lowest level of the stack is characterized by the physical resources on top of which the infrastructure is deployed. These resources can be of different nature: clusters, datacenters, and spare desktop machines. Infrastructure supporting commercial Cloud deployments are more likely to be constituted by datacenters hosting hundreds or thousands of machines, while private Clouds can provide a more heterogeneous scenario in which even the idle CPU cycles of spare desktop machines are used to leverage the compute workload. This level provides the "horse power" of the Cloud.

The physical infrastructure is managed by the core middleware layer whose objectives are to provide an appropriate run time environment for applications and to exploit at best the physical resources. In order to provide advanced services, such as application isolation, quality of service, and sandboxing, the core middleware can rely on virtualization technologies. Among the different solutions for virtualization, hardware level virtualization and programming language level virtualization are the most popular. Hardware level virtualization guarantees complete isolation of applications and a fine partitioning of the physical resources, such as memory and CPU, by means of virtual machines. Programming level virtualization provides sandboxing and managed execution for applications developed with a specific technology or programming language (i.e. Java, .NET, and Python). On top of this, the core middleware provides a wide set of services that assist service providers in delivering a professional and commercial service to end users. These services include: negotiation of the quality of service, admission control, execution management and monitoring, accounting, and billing.
Together with the physical infrastructure the core middleware represents the platform on top of which the applications are deployed in the Cloud. It is very rare to have direct access to this layer. More commonly, the services delivered by the core middleware are accessed through a user level middleware. This provides environments and tools simplifying the development and the deployment of applications in the Cloud: web 2.0 interfaces, command line tools, libraries, and programming languages. The user level middleware constitutes the access point of applications to the Cloud.

The Cloud Computing model introduces several benefits for applications and enterprises. The adaptive management of the Cloud allows applications to scale on demand according to their needs: applications can dynamically acquire more resource to host their services in order to handle peak workloads and release when the load decreases. Enterprises do not have to plan for the peak capacity anymore, but they can provision as many resources as they need, for the time they need, and when they need. Moreover, by moving their IT infrastructure into the Cloud, enterprise can reduce their administration and maintenance costs. This opportunity becomes even more appealing for startups, which can start their business with a small capital and increase their IT infrastructure as their business grows. This model is also convenient for service providers that can maximize the revenue from their physical infrastructure. Besides the most common “pay as you go” strategy more effective pricing policies can be devised according to the specific services delivered to the end user. The use of virtualization technologies allows a fine control over the resources and the services that are made available at runtime for applications. This introduces the opportunity of adopting various pricing models that can benefit either the customers or the vendors.

The model endorsed by Cloud Computing provides the capability of leveraging the execution of applications on a distributed infrastructure that, in case of public clouds, belongs to third parties. While this model is certainly convenient, it also brings additional issues from a legal and a security point of view. For example, the infrastructure constituting the Computing Cloud can be made of datacenters and clusters located in different countries where different laws for digital content apply. The same application can then be considered legal or illegal according to the where is hosted. In addition, privacy and confidentiality of data depends on the location of its storage. For example, confidentiality of accounts in a bank located in Switzerland may not be guaranteed by the use of data center located in United States. In order to address this issue some Cloud Computing vendors have included the geographic location of the hosting as a parameter of the service level agreement made with the customer. For example, Amazon EC2 provides the concept of availability zones that identify the location of the datacenters where applications are hosted. Users can have access to different availability zones and decide where to host their applications. Since Cloud Computing is still in its infancy the solutions devised to address these issues are still being explored and will definitely become fundamental when a wider adoption of this technology takes place.

Table 1 identifies some of the major players in the field and the kind of service they offer. Amazon Elastic Compute Cloud (EC2) operates at the lower levels of the Cloud Computing reference model. It provides a large computing infrastructure and a service based on hardware virtualization. By using the Amazon Web Services users can create Amazon Machine Images (AMIs) and save them as templates from which multiple instances can be run. It is possible to run either Windows or Linux virtual machines and the user is charged per hour for each of the instances running. Amazon also provides storage services with the Amazon Simple Storage Service (S3), users can
take advantage of Amazon S3 to move large data files into the infrastructure and get access to them from virtual machine instances.

Table 1. Feature comparison of some of the commercial offerings for Cloud Computing.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Amazon EC2</th>
<th>Google AppEngine</th>
<th>Microsoft Azure</th>
<th>Manjrasoft Aneka</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Type</td>
<td>IaaS</td>
<td>IaaS – PaaS</td>
<td>IaaS – PaaS</td>
<td>PaaS</td>
</tr>
<tr>
<td>Support for (value offer)</td>
<td>Compute/storage</td>
<td>Compute</td>
<td>Compute</td>
<td></td>
</tr>
<tr>
<td>Value added service provider</td>
<td>Yes</td>
<td>(web applications)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>User access interface</td>
<td>Web APIs and Command Line Tools</td>
<td>Web APIs and Command Line Tools</td>
<td>Azure Web Portal</td>
<td>Web APIs, Custom GUI</td>
</tr>
<tr>
<td>Virtualization</td>
<td>OS on Xen hypervisor</td>
<td>Application Container</td>
<td>Service Container</td>
<td>Service Container</td>
</tr>
<tr>
<td>Platform (OS &amp; runtime)</td>
<td>Linux, Windows</td>
<td>Linux</td>
<td>.NET on Windows</td>
<td>.NET/Mono on Windows, Linux, MacOS X</td>
</tr>
<tr>
<td>Deployment model</td>
<td>Customizable VM</td>
<td>Web apps (Python, Java, JRuby)</td>
<td>Azure Services</td>
<td>Applications (C#, C++, VB, ….)</td>
</tr>
<tr>
<td>If PaaS, ability to deploy on on 3rd party IaaS</td>
<td>N.A.</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

While the commercial offer of Amazon can be classified completely as a IaaS solutions, Google AppEngine, and Microsoft Azure are integrated solutions providing both a computing infrastructure and a platform for developing applications. Google AppEngine is a platform for developing scalable web applications that will be run on top of server infrastructure of Google. It provides a set of APIs and an application model that allow developers to take advantage of additional services provided by Google such as Mail, Datastore, Memcache, and others. By following the provided application model, developers can create applications in Java, Python, and JRuby. These applications will be run within a sandbox and AppEngine will take care of automatically scaling when needed. Google provides a free limited service and utilizes daily and per minute quotas to meter and price applications that require a professional service.

Azure is the solution provided by Microsoft for developing scalable applications for the Cloud. It is a cloud services operating system that serves as the development, run-time, and control environment for the Azure Services Platform. By using the Microsoft Azure SDK developers can create services that leverage on the .NET Framework. These services are then uploaded to the Microsoft Azure portal and executed on top of Windows Azure. Microsoft Azure provides additional services such as workflow execution and management, web services orchestration, and access to SQL data stores. Currently, Azure is still in Community Technical Preview and its usage is free, its commercial launch is scheduled for the second half of 2009 and users will be charged by taking into account the CPU time, the bandwidth and the storage used, the number of transaction performed by their services, and also the use of specific services such as SQL or .NET services.

Differently from all the previous solutions, Aneka is a pure implementation of the Platform as a Service model. The core value of Aneka is a service oriented runtime environment that is deployed on both physical and virtual infrastructures and allows the
execution of applications developed with different application models. Aneka provides a Software Development Kit (SDK) allowing developers to create cloud applications on any language supported by the .NET runtime and a set of tools for quickly setting up and deploying clouds on Windows and Linux based systems. Aneka can be freely downloaded and tried for a limited period, while specific arrangements have to be made with Manjrasoft for commercial use. In the remainder of this chapter we illustrate the features of Aneka.

2. Aneka Architecture

Aneka is a platform and a framework for developing distributed applications on the Cloud. It harnesses the spare CPU cycles of a heterogeneous network of desktop PCs and servers or datacenters on demand. Aneka provides developers with a rich set of APIs for transparently exploiting such resources and expressing the business logic of applications by using the preferred programming abstractions. System administrators can leverage on a collection of tools to monitor and control the deployed infrastructure. This can be a public cloud available to anyone through the Internet, or a private cloud constituted by a set of nodes with restricted access.

Aneka is based on the .NET framework and this is what makes it unique from a technology point of view as opposed to the widely available Java based solutions. While mostly designed to exploit the computing power of Windows based machines, which are most common within an enterprise environment, Aneka is portable over different platforms and operating systems by leveraging other implementations of the ECMA 334 [5] and ECMA 335 [6] specifications such as Mono. This makes Aneka an interesting solution for different types of applications in educational, academic, and commercial environments.

2.1. Overview

Figure 3 gives an overview of the features of Aneka. The Aneka based computing cloud is a collection of physical and virtualized resources connected through a network, which could be the Internet or a private intranet. Each of these resources hosts an instance of the Aneka Container representing the runtime environment in which the distributed applications are executed. The container provides the basic management features of the single node and leverages all the other operations on the services that it is hosting. In particular we can identify fabric, foundation, and execution services. Fabric services directly interact with the node through the Platform Abstraction Layer (PAL) and perform hardware profiling and dynamic resource provisioning. Foundation services identify the core system of the Aneka middleware, they provide a set of basic features on top of which each of the Aneka containers can be specialized to perform a specific set of tasks. Execution services directly deal with the scheduling and execution of applications in the Cloud. One of the key features of Aneka is the ability of providing different ways for expressing distributed applications by offering different programming models; execution services are mostly concerned with providing the middleware with an implementation for these models. Additional services such as persistence and security are transversal to the entire stack of services that are hosted by the Container. At the application level, a set of different components and tools are
provided to: 1) simplify the development of applications (SDK); 2) porting existing applications to the Cloud; and 3) monitoring and managing the Aneka Cloud.

Figure 3. Overview of the Aneka framework.

A common deployment of Aneka is presented in Figure 4. An Aneka based Cloud is constituted by a set of interconnected resources that are dynamically modified according to the user needs by using resource virtualization or by harnessing the spare CPU cycles of desktop machines. If the deployment identifies a private Cloud all the resources are in house, for example within the enterprise. This deployment is extended by adding publicly available resources on demand or by interacting with other Aneka public clouds providing computing resources connected over the Internet. The heart of this infrastructure is the Aneka Container which represents the basic deployment unit of Aneka based clouds.

Some of the most characteristic features of the Cloud Computing model are:
flexibility,
elasticity (scaling up or down on demand), and
pay per usage.

The architecture and the implementation of the Container play a key role in supporting these three features: the Aneka cloud is flexible because the collection of services available on the container can be customized and deployed according to the specific needs of the application. It is also elastic because it is possible to increase on demand the number of nodes that are part of the Aneka Cloud according to the user needs. The integration of virtual resources into the Aneka Cloud does not introduce specific challenges: once the virtual resource is acquired by Aneka it is only necessary to have an administrative account and a network access to it and deploy the Container on it as it happens for any other physical node. Moreover, because of the Container being the interface to hosting node it is easy to monitor, meter, and charge any distributed application that runs on the Aneka Cloud.

Figure 4. Deployment scenario for Aneka

2.2. Anatomy of the Aneka Container

The Container represents the basic deployment unit of Aneka based Clouds. The network of containers defining the middleware of Aneka constitutes the runtime environment hosting the execution of distributed applications. Aneka strongly relies on a Service Oriented Architecture [7] and the Container is a lightweight component...
providing basic node management features. All the other operations that are required by the middleware are implemented as services.

Figure 3 illustrates the stack of services that can be found in a common deployment of the Container. It is possible to identify four major groups of services:

- Fabric Services
- Foundation Services
- Execution Services
- Transversal Services

The collective execution of all these services actually creates the required runtime environment for executing applications. Fabric services directly interface with the hosting resource and are responsible for low level operations, foundation services constitute the core of the runtime environment, and execution services manage the execution of applications. A specific class – Transversal Services – operates at all levels and provides support for security and persistence.

Additional and specific services can be seamlessly integrated into the Container by simply updating a configuration file. This operation can be performed either by means of an automated procedure or manually. The ability of hosting on demand new services and unloading existing services makes the Aneka Container an extremely configurable component able to address and elastically react to the changing needs of the applications by scaling up or down the set of services installed in the system. Moreover, by relying on services and message passing for implementing all the features of the system, the Aneka Container can easily evolve and integrate new features with minimum setup costs.

2.3. Fabric Services

Fabric services define the lowest level of the software stack representing the Aneka Container. They provide access to the resource provisioning subsystem and to the hardware of the hosting machine. Resource provisioning services are in charge of dynamically providing new nodes on demand by relying on virtualization technologies, while hardware profile services provide a platform independent interface for collecting performance information and querying the properties of the host operating system and hardware.

Hardware profiling services provide a platform independent interface for accessing the operating system and the underlying hardware. These services rely on the Platform Abstraction Layer (PAL) that allows the Container to be completely independent from the hosting machine and the operating system and the whole framework to be portable over different platforms. In particular the following information is collected for all the supported runtimes and platforms:

- Static and dynamic CPU information (CPUs, operating frequency, CPU usage);
- Static and dynamic memory information (size, available, and used);
- Static and dynamic storage information (size, available, and used);

This information is collected for each of the nodes belonging to the Aneka Cloud and made available to the other services installed in the systems. For example,
execution services and in particular scheduling components, can take advantage of dynamic performance information to devise a more efficient scheduling for applications.

Dynamic resource provisioning allows the Aneka Cloud to elastically scale up and down according to the requirements of applications. These services are in charge of dynamically acquiring and integrating new nodes into the Aneka Cloud in order to satisfy the computation needs of one or more applications. Dynamic resource provisioning addresses two different scenarios: physical resource provisioning and virtual resource provisioning. With physical resource provisioning one Aneka Cloud simply “borrows” some nodes from other Aneka Clouds by specifying a service level agreement and the specific characteristics required for these nodes in terms of services and hardware. With virtual resource provisioning the nodes are dynamically acquired by interacting with existing virtual machine managers or IaaS implementations such as Amazon EC2 or Amazon S3. In this case, the Aneka Cloud requests as many virtual machines as needed to deploy an Aneka Container together with the required services. The way in which new resources are integrated into the Cloud characterizes the type of Cloud managed by Aneka. If resources are collected from a private internal network either via a hypervisor or another Aneka Cloud, the resulting system is still a private Cloud. If resources are obtained by relying on a publicly available Aneka Cloud, the entire system may be a public or hybrid Cloud. We have a public Cloud if the initial system was a public Cloud, a hybrid Cloud otherwise.

Resource provisioning and hardware profiling are fundamental in a Cloud environment where resources are obtained on demand and subject to specific service level agreements. In particular resource reservation strongly relies on the information obtained by these services. Aneka allows reserving nodes for a specific application. It is possible to specify the set of characteristics required for each of these nodes, and the number of nodes. The reservation service will then, if possible, reserve within the Aneka Cloud those nodes that fulfill the requirements requested by the application. To accomplish this it is necessary to access to the static and dynamic performance information of the node. Advanced strategies can then rely on dynamic resource provisioning in order to make up for the lack of resources.

2.4. Foundation Services

Together with the fabric services the foundation services represent the core of the Aneka middleware on top of which Container customization takes place. Foundation services constitute the pillars of the Aneka middleware and are mostly concerned with providing runtime support for execution services and applications. The core of Aneka addresses different issues:

- Directory and Membership;
- Resource reservation;
- Storage management;
- Licensing, accounting, and pricing;

These services can be directly consumed by users, applications, or execution services. For example, users or applications can reserve nodes for execution, while execution services can query the Membership Catalogue in order to discover whether the required services are available in the Cloud to support the execution of a specific
application. Licensing, accounting, and pricing are services that will be more of interest for single users or administrators.

2.4.1. Directory and Membership

Directory and Membership Services are responsible for setting up and maintaining the information about the nodes and the services constituting the Aneka Cloud. These services include Membership Catalogue, Heartbeat Service, and Discovery Service. The Membership Catalogue acts as a global directory maintaining the list of available services and their location in the Aneka Cloud. The information in the Membership Catalogue is dynamically updated by the Heartbeat Services installed in each node belonging to the Cloud. The Heartbeat services collect the statistic information about the hosting node from the Hardware profiling services and update the Membership Catalogue periodically. The Aneka middleware exposes some autonomic properties [8] being able not only to react to failures but also to auto-configure itself when connections between nodes are broken and nodes are not reachable. This ability is mostly provided by the Discovery Service, which is in charge of discovering the available Aneka nodes on the Cloud and providing the required information for adding a node to the Membership Catalogue. The collective execution of these three services allows the automatic setting up of an Aneka Cloud without any static configuration information, but simply an available network connection.

2.4.2. Resource Reservation

Resource reservation is a fundamental feature in any distributed middleware aiming to support application execution with a specific quality of service (QoS). Resource reservation identifies the ability of reserving a set of nodes and using them for executing a specific application. Without such capability, it is impossible to guarantee many of the most important QoS parameters, since it is not possible to control the execution of applications. Aneka provides an advanced reservation infrastructure that works across almost all the supported programming models, that allows users to reserve a collection of nodes for a given time frame, and assign this reservation to a specific application. The infrastructure guarantees that at the time specified within the reservation the selected resources are made available for executing the application.

In order to support the ability of reserving compute resources two different components have been implemented: Reservation Service and Allocation Manager. The Reservation Service is a central service that keeps track of the allocation map of all the nodes constituting the Aneka Cloud, while the Allocation Manager provides a view of the allocation map of the local Container. The Reservation Service and the Allocation Manager Services deployed in every Container provide the infrastructure that enables to reservation of compute resources, and guarantee the desired QoS. During application execution a collection of jobs are submitted to the Aneka Cloud and each of these jobs are actually moved and executed in the runtime environment set up by the Container on a specific resource. Reserved nodes only accept jobs that belong to the reservation request that is currently active. In case there is no active reservation on the node any job that matches the security requirements set by Aneka Cloud is executed. The Allocation Manager is responsible for keeping track of the reserved time frames in the local node and of checking – before the execution of jobs start – whether they are admissible or not. The Reservation Service is indeed responsible for providing a global view to the execution services and users of the status of the system, and, by
interacting with the cloud schedulers, for implementing a reservation aware application execution.

In a cloud environment, the ability of reserving resources for application execution is fundamental, not only because it offers a ways for guaranteeing the desired QoS, but also because it provides an infrastructure to implement pricing mechanisms. Aneka provides some advanced features integrated within the Reservation Service that allow a flexible pricing scheme for applications. In particular it implements the alternate offers protocol [9], which allows the infrastructure to provide the user with a counter offer in case the QoS parameters of the initial request cannot be met by the system. This feature, together with the ability of dynamically provisioning additional nodes for computation, makes the reservation infrastructure a key and innovative characteristic of Aneka.

2.4.3. Storage management

The availability of disk space, or more generally storage, is a fundamental feature for any distributed system implementation. Applications normally require files to perform their tasks, whether they are data files, configuration files, or simply executable files. In a distributed context these files have to be moved – or at least made reachable from – where the execution takes place. These tasks are normally carried out by the infrastructure representing the execution middleware and in a cloud environment these operations become even more challenging because of the dynamic nature of the system.

In order to address this issue Aneka implements a Storage Service. This service is responsible for providing persistent, robust, file based storage for applications. It constitutes a staging facility for all the nodes belonging to the Aneka Cloud and also performs data transfers among Aneka nodes, the client machine, and remote servers. In a cloud environment the user requirements can be different and dynamically change during the lifetime of the applications. Such requirements can also affect storage management in terms of their location and of the specific media used to transfer information. Aneka provides an infrastructure able to support a different set of storage facilities. The current release of Aneka provides a storage implementation based on the File Transfer Protocol (FTP) service. Additional storage facilities can be integrated into the system by providing a specific implementation of a data channel. A data channel represents the interface used within Aneka to access a specific storage facility. Each data channel implementation consists of a server component, that manages the storage space made available with the channel, and a client component, which is used to remotely access that space. Aneka can transparently plug any storage facility for which a data channel implementation has been provided and transparently use it. The use of data channels is transparent to users too, who simply specify the location of the files needed by their application and the protocol through which they are made accessible. Aneka will automatically the system with the components needed to import the required files into the Cloud.

The architecture devised to address storage needs in Aneka provides a great flexibility and extensibility. Not only different storage facilities can be integrated but they also can be composed together in order to move data across different mediums and protocols. This allows Aneka Clouds a great level of interoperability from the perspective of data.
2.4.4. Licensing, Accounting, and Pricing

Aneka provides an infrastructure that allows setting up public and private clouds. In a cloud environment, especially in the case of public clouds, it is important to implement mechanisms for controlling resources and pricing their usage in order to charge users. Licensing, accounting, and pricing are the tasks that collectively implement a pricing mechanism for applications in Aneka.

The **Licensing Service** provides the very basic resource controlling feature that protects the system from misuse. It restricts the number of resources that can be used for a certain deployment. Every container that wants to join the Aneka Cloud is subject to verification against the license installed in the system and its membership is rejected if restrictions apply. These restrictions can involve the number of maximum nodes allowed in the Aneka Cloud, or a specific set of services hosted by the container. This service does not provide any direct benefit for users but prevent the system from malicious system administrators that want to overprovision the Aneka Cloud.

The **Accounting and Pricing Services**, available in the next release of Aneka, are more directly related with billing the user for using the Cloud. In particular the Accounting Service keeps track of applications running, their reservations, and of the users they belong to, while the Pricing Service is in charge of providing flexible pricing strategies that benefit both the users of the Cloud and the service providers. These two components become important in case of dynamic resource provisioning of virtual resources: IaaS implementations such as Amazon EC2 charge the usage of the virtual machines per hour. The way in which the cost of this service is reflected into the user bill is the responsibility of the Pricing Service.

2.5. Execution Services

Execution services identify the set of services that are directly involved in the execution of distributed applications in the Aneka Cloud. The application model enforced by Aneka represents a distributed application as a collection of jobs. For any specific programming model implemented in Aneka at least two components are required providing execution support: Scheduling Service and Execution Service. The Scheduling Service coordinates the execution of applications in the Aneka Cloud and is responsible for dispatching the collection of jobs generated by applications to the compute nodes. The Execution Service constitutes the runtime environment in which jobs are executed. More precisely, it is in charge of retrieving all the files required for execution, monitoring the execution of the job, and collecting the results. The number and the type of services required to deploy a programming model varies according to the specific nature of the programming model. Generally these two services are the only ones required in most of the cases. The Task Model, the Thread Model, and the MapReduce Model are implemented according to this scheme.

Execution Services can then rely on other existing services, available with a common deployment of the Aneka Cloud, to provide a better support for application execution. For example they can integrate with the existing Reservation Service and Storage service to support quality of service for application execution and support for data transfer. The integration with these services is completely dynamic and no static binding is required.

A common deployment scenario of an Aneka Cloud concentrates the scheduling services of all the programming models in one or few nodes, while configuring all the
other nodes with execution services, thus creating a master-slave topology. Whereas this deployment is quite common, the service oriented architecture of Aneka does not enforce it and more balanced and dynamic topologies can be devised by system administrators. For example, environments characterized by thousands of machines can more easily scale and reconfigure by means of hierarchical topologies and brokering infrastructures. Hierarchical topologies can help in distributing the overload of managing huge number of resources: in this setting, the scheduling service managing a network of nodes where execution services are deployed. These scheduling services can be then seen as multi-core from other meta schedulers which coordinate the load of the system at a higher level. Such structure can be enhanced and made more dynamic by integrating into the Aneka Container brokering services that, by means of dynamic SLAs extend and enrich the set of features that are offered to the users of the Cloud. Other solutions [10], based on a peer to peer model, can also be implemented.

2.6. Transversal Services

Aneka provides additional services that affect all the layers of the software stack implemented in the Container. For this reason they are called *transversal services*, such as the persistence layer and the security infrastructure.

2.6.1. Persistence

The persistence layer provides a complete solution for recording the status of the Cloud and for restoring it after a system crash or a partial failure. The persistence layer keeps track of the sensitive information for the distributed system such as: all the applications running in the Cloud and their status; the topology information of the Cloud and the current execution status; the status of the storage. This information is constantly updated and saved to the persistence storage. The persistence layer is constituted by a collection of persistence stores that are separately configured in order to provide the best required quality of service.

The current release of Aneka provides two different implementations for these components that can be used to configure and tune the performance of the Cloud:

- **In memory persistence**: this persistence model provides a volatile store that is fast and performing but not reliable. In case of system crash or partial failure the execution of the applications can be irreversibly compromised. While this solution is optimal for a quick setup of the Aneka Cloud and for testing purposes, it is not suggested for production deployments.

- **Relational Database**: this solution relies on the ADO.NET framework for providing a persistent store, which is generally represented by a database management system. In this case the information of the state of the Cloud and its components are saved inside database tables and retrieved when necessary. This solution provides reliability against failures and prevents from the loss of data but requires an existing installation of the supported RDBMS. The current implementation of Aneka supports two different backend for this kind of solution: MySQL 5.1 and SQL Server 2005 v9.0 onward.

These are just two ready to use implementations of the persistence layer. Third parties can provide a specific implementation and seamlessly integrate it into the systems with minimum effort. The possibilities for extending the system are many: it is...
possible to implement from scratch a new persistence layer or simply provide the SQL scripts that create tables and stored procedures for the database persistence layer.

2.6.2. Security

The security layer provides access to the security infrastructure of Aneka. This layer separates authentication – that means identifying who users are – from authorization – that means what users are allowed to do. The implementation of these two functions relies on providers, which abstract the two operations within the framework, and user credentials, which contain the information required by the providers to authenticate and authorize users. Before any operation on behalf of the user is performed on the Aneka Cloud its credentials are verified against the authentication and authorization providers, which can reject or authorize the operation.

Specific implementations of these two providers can be seamlessly integrated into the infrastructure simply by editing the configuration of the Container. In this way it is possible to run Aneka on different security infrastructure according to specific requirements of the Cloud. Specific deployments can require the use of existing security infrastructures. In this case, the specific implementation of security providers will rely on the existing security model and user credentials will contain the required information for representing the user within the underlying security system. This has been the approach for supporting the Window Authentication in Aneka. In the case of Windows based deployments Aneka can rely on the Windows integrated security and provide access to the system for the specific Windows users. Alternatively, it is possible to set up a Cloud with no security at all, simply by using the Anonymous security providers, which do not perform any security check for user applications. Third parties can set up their own security providers by implementing the interfaces defined in the Aneka security APIs.

2.7. Portability and Interoperability

Aneka is a Platform as a Service implementation of the Cloud Computing model and necessarily relies on the existing virtual and physical infrastructure for providing its services. More specifically, being developed on top of the Common Language Infrastructure, it requires an implementation of the ECMA 335 specification such as the .NET framework or Mono.

Since the Cloud is a dynamic environment aggregating heterogeneous computing resources, the choice of developing the entire framework on top of a virtual runtime environment, provides some interesting advantages. For example it is possible to easily support multiple platform and operating systems with reduced or no conversion costs at all. Developing for a virtual execution environment such as Java or the Common Language Infrastructure, does not necessarily mean to devise software that will naturally run on any supported platform. In the case of Aneka this aspect becomes even more challenging since some of the components of the framework directly interact with the hardware of the machine (physical or virtual) hosting the Aneka Container.

In order to address this issue a specific layer that encapsulates all the platform dependencies on the hosting platform behind a common interface has been integrated into Aneka. This layer is called Platform Abstraction Layer (PAL) and provides a unified interface for accessing all the specific properties of the Operating System and the underlying hardware that are of interest for Aneka. The PAL is a fundamental
component of the system and constitutes the lowest layer of the software stack implemented in the Aneka Container. It exposes the following features:

- Uniform and platform independent interface for profiling the hosting platform;
- Uniform access to extended and additional properties of the hosting platform;
- Uniform and platform independent access to remote nodes;
- Uniform and platform independent management interfaces;

The dynamic and heterogeneous nature of computing clouds necessarily requires a high degree of flexibility in aggregating new resources. In the case of Aneka, adding one resource to the Cloud implies obtaining access to a physical or a virtual machine and deploying into it an instance of the Aneka Container. These operations are performed by the PAL, which not only abstracts the process for installing and deploying a new Container but also automatically configures it according to the hosting platform. At startup the container probes the system, detects the required implementation of the PAL, and loads it in memory. The configuration of the Container is performed in a completely transparent manner and makes its deployment on virtual and physical machines really straightforward.

The current release of Aneka provides a complete implementation of the PAL for Windows based systems on top of the .NET framework and for the Linux platform on top of Mono. A partial but working implementation of the PAL for Mac OS X based systems on top of Mono is also provided.

3. Application Development

Aneka is a platform for developing applications that leverage Clouds for their execution. It then provides a runtime infrastructure for creating public and private Clouds and a set of abstractions and APIs through which developers can design and implement their applications. More specifically Aneka provides developers with a set of APIs for representing the Cloud application and controlling their execution, and a set of Programming Models that are used to define the logic of the distributed application itself. These components are part of the Aneka Software Development Kit.

3.1. The Aneka SDK

The Aneka Software Development Kit contains the base class libraries that allow developers to program applications for Aneka Clouds. Beside a collection of tutorials that thoroughly explain how to develop applications, the SDK contains a collection of class libraries constituting the Aneka Application Model, and their specific implementations for the supported programming models.

The Aneka Application Model defines the properties and the requirements for distributed applications that are hosted in Aneka Clouds. Differently from other middleware implementations Aneka does not support single task execution, but any unit of user code is executed within the context of a distributed application. An application in Aneka is constituted by a collection of execution units whose nature depends on the specific programming model used. An application is the unit of deployment in Aneka and configuration and security operates at application level. Execution units constitute the logic of the applications. The way in which units are
scheduled and executed is specific to the programming model they belong to. By using this generic model, the framework provides a set of services that work across all programming model supported: storage, persistence, file management, monitoring, accounting, and security.

Figure 5. Aneka application model.

Figure 5 illustrates the key elements of the Aneka Application Model. As previously introduced an application is a collection of work units that are executed by the middleware. While the Application class contains the common operations for all the supported programming models, its template specialization customizes its behavior for
a specific model. In particular each of the programming model implementations has to specify two types: the specific type of work unit and the specific type of application manager. The work unit represents the basic unit of execution of the distributed application, while the application manager is an internal component that is used to submit the work units to the middleware. The SDK provides base class implementations for these two types and developers can easily extend them and taking advantage of the services built for them.

The Software Development Kit also provides facilities for implementing the components required by the middleware for executing a programming model. In particular, it provides some base classes that can be inherited and extended for implementing schedulers and executors components. Developers that are interested in developing a new programming model can take as a reference the existing programming models and implement new models as a variation of them or they can completely from scratch by using the base classes. Moreover, the Aneka SDK also exposes APIs for implementing custom services that can be seamlessly plugged into the Aneka Container by editing its configuration file.

3.2. Programming Models

A programming model represents a way for expressing a distributed application within Aneka. It defines the abstractions used by the user to model their application and the execution logic of these applications as a whole in the Aneka Cloud. Every application that is executed in the Aneka Cloud is expressed in terms of a specific programming model. The current release of Aneka includes three different programming models ready to use for developing applications. These are: Task Programming Model, Thread Programming Model, and MapReduce Programming Model.

3.2.1. Task Programming Model

The Task Programming Model provides developers with the ability of expressing bag of tasks applications. By using the Task Model the user can create a distributed application and submit a collection of tasks to Aneka. The submission can be either static or dynamic. The scheduling and execution services will manage the execution of these tasks according to the available resources in the Aneka network.

Developers can use predefined tasks that cover the basic functionalities available from the OS shell or define new tasks by programming their logic. With tasks being independent from each other, this programming model does not enforce any execution order or sequencing but these operations have to be completely managed by the developer on the client application if needed.

The task programming model is the most straightforward programming model available with Aneka and can be used as a base on top of which other models can be implemented. For example the parameter sweeping APIs used by the Design Explorer rely on the Task Model APIs to create and submit the tasks that are generated for each of the combinations of parameters that need to be explored. More complex models such as workflows can take advantage of this simple and thin implementation for distributing the execution of tasks.
3.2.2. Thread Programming Model

The Thread Programming Model allows quickly porting multi-threaded applications into a distributed environment. Developers familiar with threading API exposed by the .NET framework or Java can easily take advantage of the set of compute resources available with Aneka in order to improve the performance of their applications.

The Thread Model provides as fundamental component for building distributed applications the concept of distributed thread. A distributed thread exposes the same APIs of a thread in the .NET framework but is executed remotely. Developers familiar with the multi-threaded applications can create, start, join, and stop threads in the same way in which these operations are performed on local threads. Aneka will take care of distributing and coordinating the execution of these threads.

Compared to the Task Model the Thread Model provides a more complex, powerful, and lower level API. While the common usage for the Task Model is “submit and forget” – that means that users submit tasks and forget of their existence until they terminate – in the case of the Thread Model the developer is supposed to have a finer control on the single threads. This model is definitely the best option when a pre-existing multi-threaded application needs to be ported to a distributed environment for improving its performance. In this case minimal changes to the existing code have to be made to run such application by using the Thread Model.

3.2.3. MapReduce Programming Model

The MapReduce Programming Model [11] is an implementation of MapReduce [12], as proposed by Google, for .NET and integrated with Aneka. MapReduce is originated by two functions from the functional language: map and reduce. The map function processes a key/value pair to generate a set of intermediate key/value pairs, and the reduce function merges all intermediate values associated with the same intermediate key. This model is particular useful for data intensive applications.

The MapReduce Programming Model provides a set of client APIs that allow developers to specify their map and reduce functions, to locate the input data, and whether to collect the results if required. In order to execute a MapReduce application on Aneka, developers need to create a MapReduce application, configure the map and reduce components, and – as happens for any other programming model – submit the execution to Aneka.

MapReduce is good example for the flexibility of the architecture of Aneka in supporting different programming abstractions. With MapReduce the tasks are not created by the user, as with the other supported programming models, but by the MapReduce runtime itself. This peculiarity of the model is hidden within the internal implementation of MapReduce, and it is transparently supported by the infrastructure.

3.3. Extending Aneka

Aneka has been designed to support multiple programming models and its service oriented architecture allows for the integration of additional services. Adding a new programming model then becomes then as easy as integrating a set of services in the Aneka middleware. The support for a specific programming model in Aneka requires the implementation of the following components:

- Abstractions for application composition;
• Client components;
• Scheduling and execution components;

Abstractions define the user view of the programming model, while the other components are internally used by the middleware to support the execution. Aneka provides a default implementation of all these components that can be further specialized to address the specific needs of the programming model. The implementation effort required to integrate a new programming model within Aneka strictly depends on the features of the programming model itself. In order to simplify this process Aneka provides a set of services that can be reused by any model. These are application store, file transfer management, resource reservation, and authentication.

Another way of implementing a new programming model is extending one of the pre-existing models and simply adding the additional features that are required. This could be the case of a workflow implementation on top the Task Model.

3.4. Parameter Sweeping Based Applications

Aneka provides support for directly running existing application on the Cloud without the need of changing their execution logic or behavior. This opportunity can be exploited when the behavior of the application is controlled by a set of parameters representing the application input data. In this case, the most common scenario is characterized by applications that have to be run multiple times with a different set of values for these parameters. Generally, all the possible combinations of parameter values have to be explored. Aneka provides a set of APIs and tools through which it is possible to leverage multiple executions on the Aneka Cloud. These are respectively the Parameter Sweeping APIs and the Design Explorer.

The Parameter Sweeping APIs are built on top of the Task Programming Model and provide support for generating a collection of tasks that will cover all possible combinations of parameter values that are contained in a reference task. The Aneka SDK includes some ready to use task classes that provide the basic operations for composing the task template: execute an application, copy, rename, and delete a file. It also provides an interface that allows developers to create task classes supporting parameter sweeping.

The Design Explorer is a visual environment that helps users to quickly create parameter sweeping applications and run it in few steps. More precisely, the Design Explorer provides a wizard allowing users to:

• Identify the executable required to run the application;
• Define the parameters that control application execution and their domains;
• Provide the required input files for running the application;
• Define all the output files that will be produced by the application and made available to the user;
• Define the sequence of commands that compose the task template that will be run remotely;

Once the template is complete, the Design Explorer allows the user to directly run it on Aneka Clouds by using the parameter sweeping APIs. Different visualizations are provided and statistics collected by the environment in order to monitor the progress of the application.
4. Cloud Maintenance and Monitoring

Aneka provides a platform on top of which it is possible to develop applications for the Cloud. The Software Development Kit addresses all the needs from a development point of view but it is just a part of the feature set required by a Cloud Computing platform. Essential in this case is the support for monitoring, managing, maintaining, and setting up computing clouds. These operations are exposed by the management API and the Platform Abstraction Layer on top of which all the management tools and interfaces have been designed. Of a particular interest are the Management Studio and the web management interfaces.

The Management Studio is an important tool for system administrators. It is a comprehensive environment that allows them to manage every aspect of Aneka Clouds from an easy to use graphical user interface. Since Clouds are constituted of hundreds and even thousands of machines both physical and virtual, it is not possible to reach and setup each single machine by hand. Having a tool that allows remote and global management is then a basic requirement. Briefly, the set of operations that can be performed through the Management Studio are the following:

- Quick setup of computing clouds;
- Remote installation and configuration of nodes;
- Remote control of containers;
- System load monitoring and tuning.

Besides the remote control features, which dramatically simplify the management of the Cloud, it is important to notice the support for viewing the aggregate dynamic statistics of Aneka Clouds. This helps administrators to tune the overall performance of the Cloud. It is also possible to probe each single node and collect the single performance statistics: the CPU and memory load information is collected from each container and by inspecting the container configuration it is possible to identify bottlenecks in the Cloud. As the entire framework, the Management Studio has been designed to be extensible: it is possible to add new features and new services by implementing management plugins that are loaded into the environment and get access to the Cloud.

The Management Studio is not the only tool available for controlling Aneka Clouds. The framework also provides a set of web interfaces that provide a programmatic management of Aneka. Currently, only a restricted set of features – resource reservation and negotiation, task submission, and monitoring – is available through web services, while the others are still under development and testing.

5. Case Studies

Aneka has been used either in the academic field or in the industry as a middleware for Cloud Computing. In this section we will briefly present some case studies that span from the scientific research to the manufacturing and gaming industry. In all of these cases Aneka has successfully contributed to solve the scalability issues faced and to increase the performance of the applications that leverage the Cloud for their computation needs.
5.1. Scientific Research

Aneka has been used to provide support for distributed execution of evolutionary optimizers and learning classifiers. In both of the cases a significant speed up has been obtained compared to the execution on a single local machine. In both of the cases an existing legacy application has been packaged to run in a distributed environment with the aid of a small software component coordinating the distributed execution.

5.1.1. Distributed Evolutionary Optimization: EMO

EMO (Evolutionary Multi-objective Optimizer) [13] is an evolutionary optimizer based on genetic algorithms. More precisely, it is a variation of the popular NSGA-II algorithm [14] that uses the information about how individuals are connected to each other – that represents the topology of the population – to drive the evolutionary process. A distributed version of EMO has been implemented on top of Aneka to reduce the execution time of the algorithm and improve the quality of the solutions.

Genetic algorithms [15] are iterative heuristics exploiting the concepts of individual, population, and genes, to define evolving optimizers. These tune their parameters by using mutation, crossover, and mating between individuals, which represent specific points in the solution space. Genetic algorithms have a brute force approach and generally require a large number of iterations to obtain acceptable results. These requirements become even more important in the case of EMO: in order to take advantage of the topology information a large number of individuals and iterations of the algorithms are required. The search for good solutions could require hours, and in the worst case up to one day, even for benchmark problems.

In order to address this issue a distributed implementation of EMO on top of Aneka has been developed [16]. The distributed version of EMO adopts a “divide and conquer” strategy and partitions the original population of individuals into smaller populations which are used to run the EMO algorithm in parallel. At the end of each parallel evaluation the results are merged and the next iteration starts. This process is repeated for a predefined number of times.

![Figure 6. Speedup of EMO on Aneka.](image-url)
Figure 6 and Figure 7 shows the results of running the EMO optimizer on Aneka Clouds for a set of well known benchmark problems ([17] and [18]). The optimization functions used for benchmarking the distributed execution are: ZDT1 to ZDT6, and DLTZ1 to DLTZ6. For each of the optimization functions tested, the graphs respectively show the speedup and the overhead generated while running the optimizer on the Aneka Cloud. It is interesting to notice that for a small number of individual there is no advantage in leveraging the execution on Aneka Clouds. As previously introduced, one of the peculiarities of EMO is the use of topology information for driving the evolutionary process. This information becomes useful when dealing with large number of individuals, at least 1000. As shown by the graphs, the speed up is significant already for 500 individuals, while for 1000 individuals the distribution overhead is completely negligible for all the tested problems.

![Distribution Overhead](image)

**Figure 7.** Distribution overhead of EMO on Aneka.

5.1.2. Distributed Learning Classifiers for Bioinformatics: XCS

Classifier systems are software systems implementing a function that maps a given attribute set $x$ to a class $y$. In most of the cases there is no analytic expression for the mapping function. Hence, classifiers use heuristics methods to mimic expected behavior of the mapping function. In particular Learning Classifier Systems (LCS) [19] learn from the input data the structure of the mapping function and adopts genetic algorithms to evolve the set of rules that provides the best performance of the classifier. Several classifiers are derived from LCS. Among these, the eXtended Classifier System (XCS) [20] is popular for the accuracy of the classifiers obtained.

Classifier systems are compute intensive algorithms whose execution time strongly depends on the number of attributes used to classify the samples of a given dataset. Large datasets or simply small datasets with a large number of attributes cause long execution times. In the field of bioinformatics, some specific large datasets containing huge amount of information are used as databases for identifying diseases or finding interesting patterns. Within this context, learning classifiers can be applied to learn from existing classified datasets in order to evolve into classifiers that can support the
classification of unseen datasets. The drawback of this approach is that the learning process can last days and does not produce good classifiers. In this scenario the need of having a fast learning process can help bioinformatics researchers to properly tune their classifiers in a reasonable time frame.

In order to reduce the time spent in the learning process of XCS classifiers a distributed implementation based on Aneka has been provided. In particular, a distributed version of XCS has been tuned to support the diagnosis of breast cancer disease by learning from Gene Expression datasets. In order to distribute the learning process the initial dataset has been partitioned into sections that have been used to evolve into different classifiers in parallel for a predefined number of iterations. At the end of each of the iterations the results obtained from each classifier are merged according to different strategies to propagate the good results. The preliminary results have shown that the use of Aneka has contributed to reduce the execution time of the learning process to the twenty percent of the execution on a single machine.

5.2. Manufacturing and Gaming Industry

Besides the research field, Aneka has been used to support real life applications and to address scalability issues in the manufacturing and gaming industry. In particular, the load generated by the rendering of train models and the online processing of multiplayer game logs have been leveraged on a private Aneka Cloud.

5.2.1. Distributed Train Model Rendering: GoFront Group

GoFront Group is China’s premier and largest nationwide research and manufacturing group of rail electric traction equipment. Its products include high speed electric locomotives, metro cars, urban transportation vehicles, and motor train sets. The IT department of the group is responsible for providing the design and prototype of the products including the high speed electric locomotives, metro cars, urban transportation vehicles, and motor trains. The raw designs of the prototypes are required to be rendered to high quality 3D images using the Autodesk rendering software called Maya. By examining the 3D images, engineers are able to identify any potential problems from the original design and make the appropriate changes.

The creation of a design suitable for mass production can take many months or even years. The rendering of three dimensional models is one of the phases that absorb a significant amount of time since the 3D model of the train has to be rendered from different points of views and for many frames. A single frame with one camera angle defined can take up to 2 minutes to render the image. The rendering of a complete set of images from one design require three days. Moreover, this process has to be repeated every time a change is applied to the model. It is then fundamental for GoFront to reduce the rendering times, in order to be competitive and speed up the design process.

In order to face this problem, a private Aneka Cloud has been set up by using the existing desktop computers and servers available in the IT department of GoFront. Figure 8 provides an overall view of the installed system. The setup is constituted by a classic master slave configuration in which the master node concentrates the scheduling and storage facilities and thirty slave nodes are configured with execution services. The task programming model has been used to design the specific solution implemented in GoFront. A specific software tool that distributes the rendering of frames in the Aneka Cloud and composes the final rendering has been implemented to help the engineers at
GoFront. By using the software, they can select the parameters required for rendering and perform the computation on the private cloud. Figure 9 illustrates the speed up obtained by distributing the rendering phase on the Aneka Cloud, compared to the previous set up constituted by a single four-core machine. As it can be noticed, by simply using a private cloud infrastructure that harnessed on demand the spare cycles of 30 desktop machines in the department, the rendering process has been reduced from days to few hours.

![Cloud setup at GoFront](image)

**Figure 8.** Cloud setup at GoFront.

![Speed up of the rendering process](image)

**Figure 9.** Speed up of the rendering process.

5.2.2. Distributed Log Processing: TitanStrike Gaming

TitanStrike Gaming provides an online community for gamers, accessible through a web portal, where they can register their profile, select their preferred multiplayer game, and play on line matches by joining a team. The service provided by TitanStrike is not providing facilities for online gaming, but building a community around them.
where players can keep and show their statistics and challenge each other. In order to provide such services, the processing of game logs, is fundamental.

An online multiplayer game is generally characterized by a game server that controls one or more matches. While a match is running, players can join and play and the information of everything happening in the game is dumped into the game logs that are used as medium for updating the status of the local view of the game of each player. By analyzing the game logs it is then possible to build the statistics of each player. Game servers generally provide an end point that can be used to obtain the log of a specific game. A single log generates information with a low frequency since the entire process is driven by humans. But in case of a portal for gaming, where multiple games are played at the same time and many players are involved in one match, the overload generated by the processing of game logs can be huge and scalability issues arise.

In order to provide a scalable infrastructure able to support the update of statistics in real time and improve their user experience, a private Aneka Cloud has been set up and integrated into the TitanStrike portal. Figure 10 provides an overall view of the cloud setup. The role of the Aneka Cloud is to provide the horse power required to simultaneously process as many game logs as possible by distributing the log parsing among all the nodes that belong to the cloud. This solution allows TitanStrike to scale on demand when there are flash crowds generated by a huge numbers of games played at the same time.

![Figure 10. Cloud set up at TitanStrike.](image-url)
6. Conclusions and Future Directions

In this book chapter we have presented Aneka, a framework providing a platform for cloud computing applications. As discussed in the introduction there are different solutions for providing support for Cloud Computing. Aneka is an implementation of the Platform as a Service approach, which focuses on providing a set of APIs that can be used to design and implement applications for the Cloud.

The framework is based on an extensible and service oriented architecture that simplifies the deployment of clouds and their maintenance and provides a customizable environment that supports different design patterns for distributed applications. The heart of the framework is represented by the Aneka Container which is the minimum unit of deployment for Aneka Clouds and also the runtime environment for distributed applications. The container hosts a collection of services that perform all the operations required to create an execution environment for applications. They include resource reservation, storage and file management, persistence, scheduling, and execution. Moreover, services constitute the extension point of the container which can be customized to support specific needs or scenarios.

By using services different programming models have been integrated in Aneka. A programming model is a specific way of expressing the execution logic of distributed applications. It provides some familiar abstractions that developers can use to define the execution flow of applications and its component. From an implementation point of view a programming model also includes a collection of services – more precisely scheduling and execution services – that make possible its execution on top of Aneka Clouds. Aneka provides a reference model for implementing new programming models and the current release supports three different programming models: independent bag of tasks, distributed threads, and MapReduce. In order to simplify the development with Aneka a Software Development Kit contains ready to use samples, tutorials, and a full API documentation which helps starting to investigate the large range of features offered by the framework.

Aneka also provides support for deploying and managing clouds. By using the Management Studio it is possible to set up either public or private clouds, monitor their status, update their configuration, and perform the basic management operations. Moreover, a set of web interfaces allows to programmatically managing Aneka Clouds.

The flexibility of Aneka has been demonstrated by using the framework in different scenarios: from scientific research, to educational teaching, and to industry. A set of case studies representing the success stories of Aneka has been reported to demonstrate that Aneka is mature enough to address real life problems used in a commercial environment.

Aneka is under continuous development. The development team is now working on providing full support for the elastic scaling of Aneka Clouds by relying on virtualized resources. Initial tests have been successfully conducted in using Amazon EC2 as a provider of virtual resources for Aneka. This feature, and the ability of interacting with other virtual machine managers, will be included in the next version of the management APIs that will simplify and extend the set of management tasks available for Aneka Clouds.
Acknowledgements

The authors would like to thank Al Mukaddim Pathan and Dexter Duncan for their precious insights in organizing the contents of this chapter.

References


This page intentionally left blank
Chapter 5

Information Processing and Applications
Building Collaborative Applications for System-level Science

Marian BUBAK\textsuperscript{a,b}, Tomasz GUBALA\textsuperscript{b,c}, Marek KASZTELNIK\textsuperscript{c} and Maciej MALAWSKI\textsuperscript{a}

\textsuperscript{a} Institute of Computer Science AGH, Mickiewicza 30, 30-059 Kraków, Poland
\textsuperscript{b} Informatics Institute, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
\textsuperscript{c} ACC CYFRONET AGH, Kraków, Nawojki 11, 30-950 Kraków, Poland

Abstract. A novel approach to scientific investigations, besides analysis of individual phenomena, integrates different, interdisciplinary sources of knowledge about a complex system to obtain an understanding of the system as a whole. This innovative way of research called system-level science, requires advanced methods and tools for enabling collaboration of research groups. This paper presents a new approach to development and execution of collaborative applications. These applications are built as experiment plans with a notation based on the Ruby language. The virtual laboratory, which is an integrated system of dedicated tools and servers, provides a common space for planning, building, improving and performing in-silico experiments by a group of developers. The application is built with elements called gems which are available on the distributed Web- and Grid-based infrastructure. The process of application developments and the functionality of the virtual laboratory are demonstrated with a real-life example of the drug susceptibility ranking application from the HIV treatment domain.

Keywords. System-level science, e-Science, collaborative applications, virtual laboratory, ViroLab

Introduction

Recently, we observe emergence of a new approach to scientific investigations which, besides of analyses of individual phenomena, integrates different, interdisciplinary sources of knowledge about a complex system, to acquire understanding of the system as a whole. This innovative way of conducting research has recently been called system-level science [1].

Scientific investigations carried out on a holistic level require new e-research environments. Such environments aim to support integration of various sources of data and computational tools to help investigators acquire understanding of a given phenomenon through modeling and simulation processes.

Biomedicine is an important example of such a field, requiring this new approach, which, in turn, must be accompanied by adequate information technology solutions. The complexity of challenges in the biomedical research and the grow-
ing number of groups and institutions involved creates more demand from that part of science for new, collaborative environments. Since biomedicine experts and research groups do not work in separation, more and more attention and effort is devoted to collaborative, inter-laboratory projects involving data and computational resources. The computer science aspects of this research, which include virtual groups, virtual organizations built around complex in-silico experiments and electronic data stores are also representative for other fields.

An example of such a collaborative application in the virology domain, being built and used in complex simulations by many cooperating users, is drug resistance evaluation for HIV treatment [2] [3]. As the final results of this simulation is important for everyday practice of clinical virologists, there are efforts to provide it as a service via the web [4]. The ViroLab project dedicates substantial resources to deliver a decision support system to help medical doctors issue HIV drug prescriptions [5], as it develops the Drug Ranking System (DRS) [6]. Treatment decision support systems, like DRS, are used and developed by many people. There are many groups involved in HIV research and users representing various expertise levels inside these groups work to deliver a valid, reasonably complete and efficiently working solution. In turn, this objective can be achieved only if the entire endeavor is backed by a solid, innovative and well-integrated technology that is both generic enough to support users with distinct assignments, yet sufficiently focused.

In this paper we present a new approach to building and running collaborative applications and the ViroLab Virtual Laboratory [8]: a collaborative, modern platform for system-level science. The laboratory is a set of dedicated tools and servers that form a common space for planning, building, improving and performing in-silico experiments in the virology domain.

In subsequent sections we show how such a complex application as DRS for HIV treatment may be designed, prepared and deployed for use in a collaborative fashion by people of different expertise levels, working towards a common objective. The next section presents an overview of related initiatives, and it is followed by a detailed explanation of operation of the proposed solution. Next, we discuss the novelty and innovation of this solution. We conclude with a summary and plans for future research.

1. Background

The need for information technology solutions supporting system-level science is indicated in the Cover Features by I. Foster and C. Kesselman [1].

Problem-solving environments and virtual laboratories have been the subject of research and development for many years [9]. Most of them are built on top of workflow systems. The LEAD [10] project is an example of a virtual laboratory for weather prediction applications; its main modules include a portal with user interfaces, a set of dedicated, distributed Grid resources and a workflow system which allows for combining the present resources together, to define task-specific processing. An example of an experimentation space is the Kepler [11] system which provides a tool for composing application workflows (which could, in particular,
be experiments). In the MyGrid [12] environment, the Taverna system is used to compose complex experiment processes out of smaller, atomic building blocks. A rich library of those basic elements allows for great flexibility and numerous different solutions can be developed. Collaborative extensions have been provided by the MyExperiment project [13]. A recent overview of dedicated environments supporting development and execution of complex applications in biomedicine is presented in [14].

Most of problem solving environments and virtual laboratories are built on top of scientific workflow systems. The work on extension of the expressiveness of their programming models, interoperability, and on enabling access to different computing resources is still a subject of research [15]. In this paper, basing on the experience from workflow systems, we present an alternative approach to building systems supporting system-level science.

The ViroLab project [7] is developing a virtual laboratory [8] for research of infectious diseases to facilitate medical knowledge discovery and provide decision support for HIV drug resistance, and this virtual laboratory may be useful in other areas of system-level science.

To overcome the limitations of the programming methods, we have defined an experiment plan notation based on a high-level scripting language - Ruby. For easy interfacing of different technologies, we have introduced a grid object abstraction level hierarchy [16]. Each grid object class is an abstract entity which defines the operations that can be invoked from the script, each class may have multiple implementations, representing the same functionality; and an implementation may have multiple instances, running on different resources [17].

2. Drug Ranking Experiment in Virtual Laboratory

2.1. Experiment Pipeline

The process of experiment preparation in the collaborative ViroLab virtual laboratory is composed of well-defined steps (Fig. 1). At the beginning, the medical expert defines requirements for the experiment: what are its objectives, what kind of data and computation is required. Subsequently, the experiment developer, by analyzing these requirements, identifies the functional blocks that constitute the application. These computational elements of the ViroLab virtual laboratory are called gems and, in most cases, are available in the distributed Web- and Grid-based infrastructure. Otherwise, they have to be created, published and registered in the virtual laboratory, thus becoming available for other developers who may reuse them in their own experiments.

Once all required computational activities are available, an experiment plan may be created. This purposes virtual laboratory provides an expressive, easy way to use a notation based on a high-level scripting language called Ruby [18]. The experiment plan is a Ruby script. The Ruby language provides a clear syntax, a full set of control structures and, as a result, it enables expressing experiments of arbitrary complexity levels in the form of scripts.

After the script is created and it fulfills (according to the developer) all the experiment requirements, it is stored in a dedicated repository and becomes avail-
Figure 1. Experiment pipeline: consecutive steps of an experiment in the virtual laboratory.

able to other members of a given virtual organization. As a result, the scientist does not need to become familiar with scripting details, and may access the virtual laboratory through a portal as well as browse and execute the available experiments using dedicated tools [19]. During application execution, provenance data is created and stored in dedicated provenance storage. This information is used by the scientist to search for interesting data and its origins [20].

The experiment script, as released by a developer, may not be optimal or may lack some functionalities. The virtual laboratory enables the scientist to easily communicate with the developer using a dedicated tool to submit user feedback, which is then used by the developer to produce a better version of the application.

The Drug Ranking System was created as a result of the experiment pipeline described above. Interpretation of the susceptibility of the HIV virus to particular drugs involves several steps. Some of these steps have to be performed manually (a blood sample has to be taken from the patient, the genetic material from the virus has to be isolated and sequenced). Once these steps are complete, a set of valid information is placed into a database. This material provides the required input for the DRS system. Knowing the nature of the experiment, a medical expert defines its structure. A set of nucleotide sequences of the HIV virus has to be obtained. These sequences are then the subject of subtype detection algorithms and alignment processes, which create a list of mutations. This list is passed to the drug resistance expert system which returns virus-to-drug susceptibility values. When the experiment plan is defined, the developer can start searching for required gems or create them if they are not available, and implement the experiment plan.

2.2. Development and Publication of Gems

As already hinted in Section 2.1, the basic computational building blocks of experiments are called experiment gems, which follows the name introduced for Ruby libraries (Ruby gems [18]). Although in the experiment script all such gems are represented with a uniform API based on the Grid Object abstraction [21], the gems themselves may be implemented using various technologies. Such an ap-
Figure 2. Gem development process.

Approach to integration of multiple technologies was motivated by the very vivid diversity of existing Grid- and Web-based middleware systems which may be used to provide access to computation. There are standard Web services, WS-RF, distributed component frameworks such as MOCCA [22] or ProActive [23], as well as large-scale job-processing systems such as EGEE LCG/gLite [24]. The goal of the Virtual Laboratory is to support gems using all these technologies.

Before a gem can be used in Virtual laboratory experiments, it has to be prepared by a gem developer. Fig. 2 shows schematically the required steps. After the interface of the gem is defined, it must be implemented using a selected technology. For simple, stateless interaction a standard Web service is the preferred solution. If a gem requires stateful (conversational) interaction and may benefit from dynamic deployment on remote resources, then implementing it as MOCCA component may be a good choice. Otherwise, if running a gem is a CPU-intensive and time-consuming task, it may be reasonable to implement it as a standalone program, which may be submitted as a job to such Grid infrastructures as EGEE or DEISA.

Once the gem is developed, it has to be registered in the Grid Resource Registry (GRR), which is a central service of the Virtual Laboratory. GRR stores a technical description (techinfo) of each gem, including all information about the interface, implementation and details required to deploy or invoke the gem. It is possible to register gems which are published by third parties on the Web in the form of Web services: in that case it is enough to provide the WSDL file, describing the given service. Before actual registration takes place, the gem developer may write testing and debugging scripts which operate directly on the gem techinfo. Following registration in the GRR, the gem becomes visible to all experiment developers and can be shared throughout the Virtual Laboratory.

In the Drug Ranking experiment described in this paper, the gems include the Drug Resistance Service [5] and the RegaDB HIV sequence alignment and subtyping tools [25].
Figure 3. Grid Object abstraction.

2.3. Experiment Planning, Scripting and Publishing

After the requirements of the experiment are defined and the missing gems developed, installed and registered in the GRR, the developer can start creating the experiment plan. The plan links data and computation into a working application. As presented in Section 2.2, the gems can be implemented using different technologies and, consequently, the creation of an experiment that connects these technologies, becomes complicated. To hide the complexity of the underlying middleware, a high-level object-oriented API called the Grid Operation Invoker – GOI [21] has been introduced. Uniform access to computations is enabled by providing three level of resource description (Fig. 3) – Grid Object, Grid Object Implementation and Grid Object Instance. During creation of the experiment plan only the highest level is used, although, if necessary, the developer can define all the resource’s technical details using one of the lower layers. The next problem that occurs while creating the experiment plan is access to the medical data. The virtual laboratory provides a high-level, secure API that enables querying different data sources with the Data Access Client – DAC (a client of the ViroLab Data Access Service [26]).

The Experiment Planning Environment (EPE [19]) supports creation of experiment plans. EPE is an RPC application based on the Eclipse platform which offers an integrated set of tools and a dedicated editor for writing experiment plans. The Domain Ontology Store (DOS) plug-in is a graphical browser that enables discovery of semantic information about the data and computational services. The Grid Resource Registry browser (GRR-browser) plug-in allows browsing registered services, their operations, input, output parameters and the attached documentation. These two plug-ins are integrated with the EPE experiment plan editor and between them provide a powerful mechanism for data and service discovery.

The DRS experiment plan (see Fig. 4) was created using this set of tools. The developer knows that three computational services (responsible for subtyping, aligning and drug ranking) are required. Using the DOS plug-in all computational parts that return subtyped, aligned and drug-ranking results are found. Afterwards, by switching from DOS to the GRR-browser plug-in, the developer is able to see the details of the gems operations. The statements which result in the creation of selected resources, are added to the experiment plan directly.
patientID = DataRequester.new.getData("Provide patient\'s ID")
region = DataRequester.new.getData("Region (\"rt\" or \"pro\")")

nucleoDB = DACConnector.new("das",
"angelina.hlrs.de:8080/wsrf/services/DataAccessService", "", "", "")
sequences = nucleoDB.executeDistributedQuery(
"select nucleotides from nt_sequence where
patient_ii=#{patientID.to_s};")

subtypesTool = GObj.create("RegaDBSubtypesTyool")
subtypes = subtypesTool.subtype(sequences)
puts "Subtypes: #{subtypes}"

mutationsTool = GObj.create("RegaDBMutationsTool")
mutationsTool.align(sequences, region)
mutations = regaDBMutationsTool.getResult

drs = GObj.create("DrugResistanceService")
puts drs.drs("retrogram", region, 100, mutations)

Figure 4. Listing of the decision support system experiment plan.

from the browser plug-in. EPE is also integrated with the Experiment Repository version control system (based on Subversion), which facilitates collaboration between developers. As a result, many developers can work on single experiment plan, sharing it with other members of a virtual organization.

The last step in experiment plan development is to make it available to the medical expert who is the application end user. The release plug-in, integrated with EPE, simplifies the experiment plan release process. During this process a new branch in the SVN repository is created and the experiment plan is copied with a unique version number and licence file.

2.4. Execution of Experiment

Both GOI and DAC are elements of the GridSpace engine (GSEngine [27]) which provides runtime support. It allows executing experiment plans locally on the developer’s machine, or remotely, on the server (Fig. 5). EPE is integrated with the runtime, thus making experiment plan creation and testing easy. For the medical expert who is the end user of the created experiments, a dedicated Web based application (Experiment Management Environment – EMI [19]) is created, hiding the complexity of the technology layer. It allows browsing information about the released experiment plans’ versions (their names, descriptions, licences) and executes them. Thanks to close integration with the GSEngine, interaction between users and experiment plans is realized. This mechanism allows receiving additional information from the user during script execution. For example, the DRS experiment (Fig. 4) requires two pieces of input data from the user: patientID – necessary to receive patient sequences from the medical database, and the region – required by the Drug Resistance Service.
3. Application examples

ViroLab virtual laboratory has been applied to a number of application domains, beyond the virology domain. Examples of such applications include:

- **Protein folding and structure comparison** The experiments in Virtual laboratory can be used to combine various models of protein folding and then to compare their output. For the comparison of nucleotide and protein sequences the alignment tools such as ClustalW can be used. All these tools can be easily and flexibly integrated using the experiment notation.

- **Data mining** We have integrated Weka datamining library with the virtual laboratory using MOCCA [22] components as the underlying technology. Experiments can combine such tools as data retrievers, classifiers, association rules generators or clusterers, which can be connected in customizable workflows and executed on the available computing resources.

- **Pocket detection in proteins** Multiple tools such as Pocket-Finder were made available for researchers to conduct series of experiments involving analysis of these important forms in protein structure. Virtual laboratory enabled automating the comparison of multiple detection algorithms on a large number of proteins.

- **Computational chemistry** The environment was used to setup and run series of Gaussian application executions. The core elements of Virtual laboratory propel the EGEE job submission and coordination of application runs while the web presentation part serves as a base for visual experiment setup and results management (a part of future computational chemistry portal).

- **Demonstration and teaching experiments** Virtual laboratory was also used for educational purposes in computer science classes. Students contributed their scientific computing projects which have been integrated with the virtual laboratory and can be combined together to demonstrate such methods as Monte-Carlo integration, PDE solving and generation of graphs and figures. The results can be shared on the Web, which makes a nice collaborative experience.
4. Innovation

The ViroLab virtual laboratory provides an environment to collaboratively plan, develop and use biomedical applications. The main innovation of the presented platform is dedication to multi-expertise task-oriented groups. Tools are provided for technical personnel, developers and administrators whose task is to maintain and enrich the experiment space. Additionally, there are tools that help virologists and healthcare providers perform their treatment-related tasks. The respective objectives and actions of these user groups are combined together with a set of remote services, information stores and other integration techniques. In this way the laboratory helps entire research teams (both traditional and virtual, Internet-wide ones) reach their scientific and professional goals more effectively.

Another innovative feature of the presented solution is stress on the generality of provided solutions in the middleware layer. The GridSpace runtime components are designed to support various remote computation technologies, programming models and paradigms. Together with this generic and multi-purpose solution, the environment provides a set of user-oriented tools that allow customizing, arranging and populating the virtual laboratory space with content and solutions specific to certain application domains. It is a method of harvesting the end users’ high creativity to help them co-create their environment rather than tailoring ready-to-use solutions. Since the e-Science domain is evolving very quickly, we argue that this model of a generic platform with specific content is best suited for technically knowledgeable teams of scientists. The described concept of independent analysis gems and data sources as well as the scripting glue used to combine them in desired experiments, ensures easy reconfigurability, extensibility and enables ad-hoc recomposition of laboratory content and applications.

The presented platform facilitates fast, close cooperation of developers and users on experiments. Since an in-silico experiment is subject to frequent changes, modifications and enhancements, the traditional software model of releases, downloads, deployments and bug reports is not effective enough. Instead, the ViroLab experiment planning and publishing model encourages quick, agile software releasing and a corresponding versioning scheme. In this model, enhancement reports can be provided right away in the experiment execution tool and they are immediately visible to all interested programmers, who may publish new experiment versions which are also immediately ready to use by all interested scientists in the group. The additional licensing and terms-of-use information, always attached to experiments, saves the end users time that would otherwise be spent on finding out whether and how the results of experiments may be used and published.

The provenance approach in the ViroLab virtual laboratory brings together ontology-based semantic modeling, monitoring of applications and the runtime infrastructure, and database technologies, in order to collect rich information concerning the execution of experiments, represent it in a meaningful way, and store it in a scalable repository [28].
5. Summary and future work

The applicability and suitability of this new approach to development and running collaborative applications as well as the virtual laboratory was demonstrated with the real-life example of the drug susceptibility ranking application from the HIV treatment domain. The novel design of the virtual laboratory allows for truly collaborative planning, development, preparation and execution of complex data acquisition and analysis applications, being so crucial for the biomedicine field. In the proposed environment people of different occupations, both advanced script developers and scientists can effectively and collaboratively conduct their respective tasks, contributing to a common goal. The current version of the presented platform, rich documentation and tutorials are available from the ViroLab virtual laboratory site [8].

In the ViroLab project, this virtual laboratory is used to plan and execute important virological experiments, with various types of analysis of the HIV virus genotype, such as the calculation of drug resistance, querying historical and provenance information about experiments, a drug resistance system based on the Retrogram ruleset. It has also been applied to other application domains, such as: protein folding and structural comparison, data mining using the Weka library, computational chemistry to develop a series of Gaussian application on the EGEE infrastructure, as an education tool in computer science classes.

We have developed an environment for collaborative planning, development and execution of e-Science applications. It facilitates fast, close cooperation of developers and users so it may be used by groups of experts running complex computer simulations. In-silico experiments undergo frequent changes, modifications and enhancements, and this platform encourages quick, agile simulation software releasing.

The laboratory is under continuous development. One of the most important features to be added is a module for management of results produced by experiments. Effort is being invested in semantic descriptions of data and computations. Consequently, finding interesting information will become easier and the corresponding middleware will be able to track the provenance of results in an application-specific way. This, in turn, will lead to future experiment repeatability. Further work also involves development of an introspection mechanism that will enable interactive execution of scripts. This is necessary for collaborative and exploratory programming and will allow the user to immediately react to the results of each simulation step.

All the above listed new functionality aspects are of great importance for system-level science.

Acknowledgements. This work was partially funded by the European Commission under the ViroLab IST-027446, the related Polish SPUB-M grant, the AGH grant 11.11.120.777, and ACC Cyfronet AGH grant 500-08. The Authors are grateful to Peter M.A. Sloot for many helpful discussions, to Piotr Nowakowski for his comments, and to Lucio Grandinetti for the opportunity to present and discuss this new approach at the HPC 2008 Conference in Centraro.
References

[8] Virolab Virtual Laboratory; http://virolab.cyfronet.pl
and Tools. Springer-Verlag (January 2006)

http://glite.web.cern.ch/glite


Parallel Data Mining from Multicore to Cloudy Grids

Geoffrey FOX\textsuperscript{a,b,1}, Seung-Hee BAE\textsuperscript{b}, Jaliya EKANAYAKE\textsuperscript{b}, Xiaohong QIU\textsuperscript{c}, and Huapeng YUAN\textsuperscript{b}

\textsuperscript{a} Informatics Department, Indiana University 919 E. 10th Street Bloomington, IN 47408 USA
\textsuperscript{b} Computer Science Department and Community Grids Laboratory, Indiana University 501 N. Morton St., Suite 224, Bloomington IN 47404 USA
\textsuperscript{c} UITS Research Technologies, Indiana University, 501 N. Morton St., Suite 211, Bloomington, IN 47404

Abstract. We describe a suite of data mining tools that cover clustering, information retrieval and the mapping of high dimensional data to low dimensions for visualization. Preliminary applications are given to particle physics, bioinformatics and medical informatics. The data vary in dimension from low (2-20), high (thousands) to undefined sequences with dissimilarities but not vectors defined. We use deterministic annealing to provide more robust algorithms that are relatively insensitive to local minima. We discuss the algorithm structure and their mapping to parallel architectures of different types and look at the performance of the algorithms on three classes of system; multicore, cluster and Grid using a MapReduce style algorithm. Each approach is suitable in different application scenarios. We stress that data analysis/mining of large datasets can be a supercomputer application.

Keywords. MPI, MapReduce, CCR, Performance, Clustering, Multidimensional Scaling

Introduction

Computation and data intensive scientific data analyses are increasingly prevalent. In the near future, data volumes processed by many applications will routinely cross the peta-scale threshold, which would in turn increase the computational requirements. Efficient parallel/concurrent algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such scientific data analyses. Most of these analyses can be thought of as a Single Program Multiple Data (SPMD) [1] algorithms or a collection thereof. These SPMDs can be implemented using different parallelization techniques such as threads, MPI [2], MapReduce [3], and mash-up [4] or workflow technologies [5] yielding different performance and usability characteristics. In some fields like particle physics, parallel data analysis is already commonplace and indeed essential. In others such as biology, data volumes are still such that much of the work can be performed on sequential machines linked together by workflow systems such as Taverna [6]. The parallelism currently exploited is usually the “almost embarrassingly parallel” style illustrated by the independent events in particle physics or the independent documents of information retrieval – these lead

\textsuperscript{1} Corresponding Author: Geoffrey Fox; E-mail: gcf@grids.ucs.indiana.edu
to independent “maps” (processing) which are followed by a reduction to give
histograms in particle physics or aggregated queries in web searches. MapReduce is a
cloud technology that was developed from the data analysis model of the information
retrieval field and here we combine this cloud technique with traditional parallel
computing ideas. The excellent quality of service (QoS) and ease of programming
provided by the MapReduce programming model is attractive for this type of data
processing problem. However, the architectural and performance limitations of the
current MapReduce architectures make their use questionable for many applications.
These include many machine learning algorithms [7, 8] such as those discussed in this
paper which need iterative closely coupled computations. Our results find poor results
for MapReduce on many traditional parallel applications with an iterative structure in
disagreement with earlier papers [7]. In section 2 we compare various versions of this
data intensive programming model with other implementations for both closely and
loosely coupled problems. However, the more general workflow or dataflow paradigm
(which is seen in Dryad [9] and other MapReduce extensions) is always valuable. In
sections 3 and 4 we turn to some data mining algorithms that require parallel
implementations for large data sets; interesting both sections see algorithms that scale
like N^2 (N is dataset size) and use full matrix operations.

<table>
<thead>
<tr>
<th>Ref</th>
<th>Cluster Name</th>
<th># Nodes</th>
<th>CPU</th>
<th>L2 Cache Memory</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Barcelona</td>
<td>1</td>
<td>1 AMD Quad Core Opteron 2356 2.3GHz</td>
<td>2x1MB 8 GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
<tr>
<td></td>
<td>(4 core Head Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Barcelona</td>
<td>4</td>
<td>2 AMD Quad Core Opteron 2356 2.3GHz</td>
<td>4×512K 16GB</td>
<td>Windows Server 2003 Enterprise x64 bit Edition</td>
</tr>
<tr>
<td></td>
<td>(8 core Compute Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Barcelona</td>
<td>2</td>
<td>4 AMD Quad Core Opteron 8356 2.3GHz</td>
<td>4×512K 16GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
<tr>
<td></td>
<td>(16 core Compute Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Barcelona</td>
<td>1</td>
<td>4 Intel Six Core Xeon E7450 2.4GHz</td>
<td>12 M 48GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
<tr>
<td></td>
<td>(24 core Compute Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Madrid</td>
<td>1</td>
<td>1 AMD Quad Core Opteron 2356 2.3GHz</td>
<td>2x1MB 8 GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
<tr>
<td></td>
<td>(4 core Head Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Madrid</td>
<td>8 (128 cores)</td>
<td>4 AMD Quad Core Opteron 8356 2.3GHz</td>
<td>4x512K 16 GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
<tr>
<td></td>
<td>(16 core Compute Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Gridfarm</td>
<td>8</td>
<td>2 Quad core Intel Xeon E5345 2.3GHz</td>
<td>4x1MB 8GB</td>
<td>Red Hat Enterprise Linux 4</td>
</tr>
<tr>
<td>H</td>
<td>IU Quarry</td>
<td>112</td>
<td>2 Quad-core Intel Xeon 5335 2.00GHz</td>
<td>4x4MB, 8GB</td>
<td>Red Hat Enterprise Linux 4</td>
</tr>
<tr>
<td></td>
<td>(24 core Compute Node)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Tempest (24 core Infiniband)</td>
<td>32 (768 cores)</td>
<td>4 Intel Six Core Xeon E7450 2.4GHz</td>
<td>12 M 48 GB</td>
<td>Windows Server HPC Edition (Service Pack 1)</td>
</tr>
</tbody>
</table>
Our algorithms are parallel MDS (Multi dimensional scaling) [10] and clustering. The latter has been discussed earlier by us [11-15] but here we extend our results to larger systems – single workstations with 16 and 24 cores and a 128 core (8 nodes with 16 cores each) cluster described in table 1. Further we study a significantly different clustering approach that only uses pairwise distances (dissimilarities between points) and so can be applied to cases where vectors are not easily available. This is common in biology where sequences can have mutual distances determined by BLAST like algorithms but will often not have a vector representation. Our MDS algorithm also only uses pairwise distances and so it and the new clustering method can be applied broadly. Both our original vector-based (VECDA) and the new pairwise distance (PWDA) clustering algorithms use deterministic annealing to obtain robust results. VECDA was introduced by Rose and Fox almost 20 years ago [16] and has obtained good results [17] and there is no clearly better clustering approach. The pairwise extension PWDA was developed by Hofmann and Buhmann [18] around 10 years ago but does not seem to have used in spite of its attractive features – robustness and applicability to data without vector representation. We complete the algorithm and present a parallel implementation in this paper.

As seen in table 1, we use both Linux and Windows platforms in our multicore and our work uses a mix of C#, C++ and Java. Our results study three variants of MapReduce, threads and MPI. The algorithms are applied across a mix of paradigms to study the different performance characteristics.

1. Choices in Messaging Runtime

The focus of this paper will be comparison of runtime environments for both parallel and distributed systems. There are successful workflow languages which underlies the approach of the SALSA project [15] which is to use workflow technologies – defined as orchestration languages for distributed computing for the coarse grain functional components of parallel computing with dedicated low level direct parallelism of kernels. At the run time level, there is much similarity between parallel and distributed run times to the extent that both support messaging but with different properties. Some of the choices are shown in figure 1 and differ by both hardware and programming models. The hardware support of parallelism/concurrency varies from shared memory multicore, closely coupled (e.g. Infiniband...
connected) clusters, and the higher latency and possibly lower bandwidth distributed systems. The coordination (communication and synchronization) of the different execution units vary from threads (with shared memory on cores); MPI between cores or nodes of a cluster; workflow or mash-ups linking services together; the new generation of cloud data intensive programming systems typified by Hadoop [19] (implementing MapReduce) and Dryad. These can be considered as the workflow systems of the information retrieval industry but are of general interest as they support parallel analysis of large datasets. As illustrated in the figure the execution units vary from threads to processes and can be short running or long lived.

![Figure 1(b)](image)

**Figure 1(b).** Last four of seven different combinations of processes/threads and intercommunication mechanisms discussed in the text

Short running threads can be spawned up in the context of persistent data in memory and so have modest overhead seen in section 4. Short running processes in the spirit of stateless services are seen in Dryad and Hadoop and due to the distributed memory can have substantially higher overhead than long running processes which are coordinated by rendezvous messaging as later do not need to communicate large amounts of data – just the smaller change information needed. The importance of this is emphasized in figure 2 showing data intensive processing passing through multiple “map” (each map is for example a particular data analysis or filtering operation) and “reduce” operations that gather together the results of different map instances corresponding typically to a data parallel break up of an algorithm. The figure notes two important patterns

![Figure 2](image)
a) **Iteration** where results of one stage are iterated many times. This is seen in the “Expectation Maximization” EM steps in the later sections where for clustering and MDS, thousands of iterations are needed. This is typical of most MPI style algorithms.

b) **Pipelining** where results of one stage are forwarded to another; this is functional parallelism typical of workflow applications. In applications of this paper we implement a three stage pipeline:

Data (from disk) → Clustering → Dimension Reduction (MDS) → Visualization

Each of the first two stages is parallel and one can break up the compute and reduce modules of figure 2 into parallel components as shown in figure 3. There is an important ambiguity in parallel/distributed programming models/runtimes that both the parallel MPI style parallelism and the distributed Hadoop/Dryad/Workflow models are implemented by messaging. Thus the same software can in fact be used for all the decompositions seen in figures 1-3. Thread coordination can avoid messaging but even here messaging can be attractive as it avoids many of the error scenarios seen in shared memory thread synchronization. The CCR threading [8-11, 20-21] used in this paper is coordinated by reading and writing messages to ports. As a further example of runtimes crossing different application characteristics, MPI has often been used in Grid (distributed) applications with MPICH-G popular here. Again the paper of Chu [7] noted that the MapReduce approach can be used in many machine learning algorithms and one of our data mining algorithms VECDA only uses map and reduce operations (it does not need send or receive MPI operations). We will show in this paper that MPI gives excellent performance and ease of programming for MapReduce as it has elegant support for general reductions although it does not have the fault tolerance and flexibility of Hadoop or Dryad. Further MPI is designed for the “owner-computes” rule of SPMD – if a given datum is stored in a compute node’s memory, that node’s CPU computes (evolves or analyzes) it. Hadoop and Dryad combine this idea with the notion of “taking the computing to the data”. This leads to the generalized “owner stores and computes” rule or crudely that a file (disk or database) is assigned a compute node that analyzes (in parallel with nodes assigned different files) the data on its file. Future scientific programming models must clearly capture this concept.

## 2. Data Intensive Workflow Paradigms

In this section, we will present an architecture and a prototype implementation of a new programming model that can be applied to most composable class of applications with various program/data flow models, by combining the MapReduce and data streaming techniques and compare its performance with other parallel programming runtimes such as MPI, and the cloud technologies Hadoop and Dryad.

MapReduce is a parallel programming technique derived from the functional programming concepts and proposed by Google for large-scale data processing in a
distributed computing environment. The map and reduce programming constructs offered by MapReduce model is a limited subset of programming constructs provided by the classical distributed parallel programming models such as MPI. However, our current experimental results highlight that many problems can be implemented using MapReduce style by adopting slightly different parallel algorithms compared to the algorithms used in MPI, yet achieve similar performance to MPI for appropriately large problems. A major advantage of the MapReduce programming model is that the easiness in providing various quality of services. Google and Hadoop both provide MapReduce runtimes with fault tolerance and dynamic flexibility support.

Dryad is a distributed execution engine for coarse grain data parallel applications. It combines the MapReduce programming style with dataflow graphs to solve the computation tasks. Dryad considers computation tasks as directed acyclic graph (DAG)s where the vertices represent computation tasks –typically, sequential programs with no thread creation or locking, and the edges as communication channels over which the data flow from one vertex to another.

Moving computation to data is another advantage of the MapReduce and Dryad have over the other parallel programming runtimes. With the ever-increasing requirement of processing large volumes of data, we believe that this approach has a greater impact on the usability of the parallel programming runtimes in the future.

2.1. Current MapReduce Implementations

Google's MapReduce implementation is coupled with a distributed file system named Google File System (GFS) [22] where it reads the data for MapReduce computations and stores the results. According to the seminal paper by J. Dean et al.[3], in their MapReduce implementation, the intermediate data are first written to the local files and then accessed by the reduce tasks. The same architecture is adopted by the Apache's MapReduce implementation – Hadoop.

Hadoop stores the intermediate results of the computations in local disks, where the computation tasks are executed, and informs the appropriate workers to retrieve (pull) them for further processing. The same approach is adopted by Disco [23] – another open source MapReduce runtime developed using a functional programming language named Erlang [24]. Although this strategy of writing intermediate result to the file system makes the above runtimes robust, it introduces an additional step and a considerable communication overhead to the MapReduce computation, which could be a limiting factor for some MapReduce computations. Apart from the above, all these runtimes focus mainly on computations that utilize a single map/reduce computational unit. Iterative MapReduce computations are not well supported.

2.2. CGL-MapReduce

CGL-MapReduce is a novel MapReduce runtime that uses streaming for all the communications, which eliminates the overheads associated with communicating via a file system. The use of streaming enables the CGL-MapReduce to send the intermediate results directly from its producers to its consumers.

Currently, we have not integrated a distributed file system such as HDFS with CGL-MapReduce, but it can read data from a typical distributed file system such as NFS or from local disks of compute nodes of a cluster with the help of a meta-data file. The fault tolerance support for the CGL-MapReduce will harness the reliable delivery
mechanisms of the content dissemination network that we use. Figure 4 shows the main components of the CGL-MapReduce.

The CGL MapReduce runtime system is comprised of a set of workers, which perform map and reduce tasks and a content dissemination network that handles all the underlying communications. As in other MapReduce runtimes, a master worker (MRDriver) controls the other workers according to instructions given by the user program. However, unlike typical MapReduce runtimes, CGL-MapReduce supports both single-step and iterative MapReduce computations.

Figure 4: Components of the CGL-MapReduce System

A MapReduce computation under CGL-MapReduce passes through several phases of computations as shown in figure 5. In CGL-MapReduce the initialization phase is used to configure both the map/reduce tasks and can be used to load any fixed data necessary for the map/reduce tasks. The map and reduce stages perform the necessary data processing while the framework directly transfers the intermediate result from map tasks to the reduce tasks. The merge phase is another form of reduction which is used to collect the results of the reduce stage to a single value. The User Program has access to the results of the merge operation. In the case of iterative MapReduce computations, the user program can call for another iteration of MapReduce by looking at the result of the merge operation and the framework performs another iteration of MapReduce using the already configured map/reduce tasks eliminating the necessity of configuring map/reduce tasks again and again as it is done in Hadoop.

CGL-MapReduce is implemented in Java and utilizes NaradaBrokering[25], a streaming-based content dissemination network. The CGL-MapReduce research prototype provides the runtime capabilities of executing MapReduce computations
written in the Java language. MapReduce tasks written in other programming languages require wrapper map and reduce tasks in order for them to be executed using CGL-MapReduce.

2.3. Performance Evaluation

To evaluate the different runtimes for their performance we have selected several data analysis applications. First, we applied the MapReduce technique to parallelize a High Energy Physics (HEP) data analysis application and implemented it using Hadoop, CGL-MapReduce, and Dryad (Note: The academic release of Dryad only exposes the DryadLINQ [26] API for programmers. Therefore, all our implementations are written using DryadLINQ although the underlying runtime it uses is Dryad). The HEP data analysis application processes large volumes of data and performs a histogramming operation on a collection of event files produced by HEP experiments. Next, we applied the MapReduce technique to parallelize a Kmeans clustering [27] algorithm and implemented it using Hadoop, CGL-MapReduce, and Dryad. Details of these applications and the challenges we faced in implementing them can be found in [28]. In addition, we implemented the same Kmeans algorithm using MPI (C++) as well. We have also implemented a matrix multiplication algorithm using Hadoop and CGL-MapReduce. We also implemented two common text-processing applications, which perform a “word histogramming” operation, and a “distributed grep” operation using Dryad, Hadoop, and CGL-MapReduce. Table 1 and Table 2 highlight the details of the hardware and software configurations and the various test configurations that we used for our evaluations.

Table 2. Test configurations.

<table>
<thead>
<tr>
<th>Feature</th>
<th>HEP Data Analysis</th>
<th>Kmeans clustering</th>
<th>Matrix Multiplication</th>
<th>Histogramming &amp; Grep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster Ref</td>
<td>H</td>
<td>G</td>
<td>G</td>
<td>B</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>12</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>96</td>
<td>32</td>
<td>40</td>
<td>32</td>
</tr>
<tr>
<td>Amount of Data</td>
<td>Up to 1TB of HEP data</td>
<td>Up to 10 million data points</td>
<td>Up to 16000 rows and columns</td>
<td>100GB of text data</td>
</tr>
<tr>
<td>Data Location</td>
<td>IU Data Capacitor: a high-speed and high-bandwidth storage system running the Lustre File System</td>
<td>Hadoop : HDFS CGL-MapReduce : NFS Dryad : Local Disc</td>
<td>Hadoop : HDFS CGL-MapReduce : NFS</td>
<td>Hadoop : HDFS CGL-MapReduce: Local Disc Dryad : Local Disc</td>
</tr>
<tr>
<td>Language</td>
<td>Java, C++ (ROOT)</td>
<td>Java, C++</td>
<td>Java</td>
<td>Java, C#</td>
</tr>
</tbody>
</table>

For the HEP data analysis, we measured the total execution time it takes to process the data under different implementations by increasing the amount of data. Figure 6 (a) depicts our results.

Hadoop and CGL-MapReduce both show similar performance. The amount of data accessed in each analysis is extremely large and hence the performance is limited by the I/O bandwidth of a given node rather than the total processor cores. The overhead induced by the MapReduce implementations has negligible effect on the overall computation.
The Dryad cluster (Table 1 ref. B) we used has a smaller hard disks compared to the other clusters we use. Therefore, to compare the performance of Hadoop, CGL-MapReduce, and Dryad for HEP data analysis, we have performed another test using a smaller data set on a smaller cluster configuration. Since Dryad is deployed on a Windows cluster running HPC Server Operating System(OS) while Hadoop and CGL-MapReduce are run on Linux clusters, we normalized the results of the this benchmark to eliminate the differences caused by the hardware and the different OSs. Figure 6(b) shows our results.

Figure 6(a). HEP data analysis, execution time vs. the volume of data (fixed compute resources)

The Dryad cluster (Table 1 ref. B) we used has a smaller hard disks compared to the other clusters we use. Therefore, to compare the performance of Hadoop, CGL-MapReduce, and Dryad for HEP data analysis, we have performed another test using a smaller data set on a smaller cluster configuration. Since Dryad is deployed on a Windows cluster running HPC Server Operating System(OS) while Hadoop and CGL-MapReduce are run on Linux clusters, we normalized the results of the this benchmark to eliminate the differences caused by the hardware and the different OSs. Figure 6(b) shows our results.

Figure 6(b). HEP data analysis, execution time vs. the volume of data (fixed compute resources). Note: In the Dryad version of HEP data analysis the “reduction” phase (combining of partial histograms produced by the “map” tasks) is performed by the GUI using a separate thread. So the timing results for Dryad does not contain the time for combining partial histograms.

Figure 6(a) and 6(b) show that Hadoop, Dryad, and CGL-MapReduce all perform nearly equally for the HEP data analysis. HEP data analysis is both compute and data intensive and hence the overheads associated with different parallel runtimes have negligible effect on the overall performance of the data analysis.
We evaluate the performance of different implementations for the Kmeans clustering application and calculated the parallel overhead ($\phi$) induced by the different parallel programming runtime using the formula given below. In this formula $P$ denotes the number of hardware processing units (i.e. number of cores used) and $T(P)$ denotes the total execution time of the program when $P$ processing units are used. $T(1)$ denotes the total execution time for a single threaded program. Note $\phi$ is just $(1/\text{efficiency} - 1)$ and often is preferable to efficiency as overheads are summed linearly in $\phi$.

$$\phi(P) = \frac{[PT(P) - T(1)]}{T(1)}$$ (2.1)

Figure 7 depicts our performance results for Kmeans expressed as overhead.

Figure 7. Overheads associated with Hadoop, Dryad, CGL-MapReduce, and MPI for Kmeans clustering – iterative MapReduce - (Both axes are in log scale)

The results in figure 7 show that although the overheads of different parallel runtimes reduce with the increase in the number of data points, both Hadoop and Dryad have very large overheads for the Kmeans clustering application compared to MPI and CGL-MapReduce implementations.

Matrix multiplication is another iterative algorithm that we have implemented using Hadoop and CGL-MapReduce. To implement matrix multiplication using MapReduce model, we adopted the row/column decomposition approach to split the matrices. To clarify our algorithm let’s consider an example where two input matrices $A$ and $B$ produce matrix $C$ as the result of the multiplication process. We split the matrix $B$ into $n$ column blocks where $n$ is equal to the number of map tasks used for the computation. The matrix $A$ is split to $m$ row blocks where $m$ determines the number of iterations of MapReduce computations needed to perform the entire matrix multiplication. In each iteration, all the map tasks consume two inputs; (i) a column block of matrix $B$ and (ii) a row block of matrix $A$ and collectively they produce a row block of the resultant matrix $C$. The column block associated with a particular map task is fixed throughout the computation while the row blocks are changed in each iteration. However, in Hadoop’s programming model, there is no way to specify this behavior and hence it
loads both the column block and the row block in each iteration of the computation. CGL-MapReduce supports the notion of long running map/reduce tasks where these task are allowed to retain static data in memory across invocations yielding better performance characteristics for iterative MapReduce computations.

For the matrix multiplication program, we measured the total execution time by increasing the size of the matrices used for the multiplication, using both Hadoop and CGL-MapReduce implementations. The result of this evaluation is shown in figure 8.

![Figure 8. Performance of the Hadoop and CGL-MapReduce for matrix multiplication](image)

The results in figure 7 and figure 8 show how the approach of configuring once and re-using of map/reduce tasks across iterations and the use of streaming have improved the performance of CGL-MapReduce for iterative MapReduce tasks. The communication overhead and the loading of static data in each iteration have resulted large overheads in iterative MapReduce computations implemented using Hadoop. The DAG based execution model of Dryad requires generation of execution graphs with fixed number of iterations. It also supports “loop unrolling” where a fixed number of iterations are performed as a single execution graph (a single query of DryadLINQ). The number of loops that can be unrolled is limited by the amount of stack space available for a process, which executes a collection of graph vertices as a single operation. Therefore, an application, which requires $n$ iterations of MapReduce computations, can perform it in $m$ cycles where in each cycle; Dryad executes a computation graph with $n/m$ iterations. In each cycle the result computed so far is written to the disk and loaded back at the next cycle. Our results show that even with this approach there are considerable overheads for iterative computations implemented using Dryad.

The performance results of the two text processing applications comparing Hadoop, CGL-MapReduce, and Dryad are shown in figure 9 and figure 10.
In both these tests, Hadoop shows higher overall processing time compared to Dryad and CGL-MapReduce. This could be mainly due to its distributed file system and the file based communication mechanism. Dryad uses in memory data transfer for intra-node data transfers and a file based communication mechanism for inter-node data transfers where as in CGL-MapReduce all data transfers occur via streaming. The “word histogramming” operation requires higher data transfer requirements compared to the “distributed grep” operation and hence the streaming data transfer approach adopted by the CGL-MapReduce shows lowest execution times for the “word histogramming” operation. In “distributed grep” operation both Dryad and CGL-MapReduce show close performance results.
3. Multidimensional Scaling MDS

Dimension reduction algorithms are used to reduce dimensionality of high dimensional data into Euclidean low dimensional space, so that dimension reduction algorithms are used as visualization tools. Some dimension reduction approaches, such as generative topographic mapping (GTM) [29] and Self-Organizing Map (SOM) [30], seek to preserve topological properties of given data rather than proximity information. On the other hand, multidimensional scaling (MDS) [31-32] tries to maintain dissimilarity information between mapping points as much as possible. The MDS algorithm involves several full $N \times N$ matrices where we are mapping $N$ data points. Thus, the matrices could be very large for large problems ($N$ could be as big as millions even today). For large problems, we will initially cluster the given data and use the cluster centers to reduce the problem size. Here we parallelize an elegant algorithm for computing MDS solution, named SMACOF (Scaling by MAjorizing a COmplicated Function) [33-34], using MPI.NET [35-36] which is an implementation of message passing interface (MPI) for C# language and presents performance analysis of the parallel implementation of SMACOF on multicore cluster systems. We show some examples of the use of MDS to visualize the results of the clustering algorithms of section 4 in figure 11. These are datasets in high dimension (from 20 in figure 11(right) to over a thousand in figure 11(left)) which are projected to 3D using proximity (distance/dissimilarity) information. The figure shows 2D projections determined by us from rotating 3D MDS results.

![Figure 11. Visualization of MDS projections using parallel SMACOF described in section 3. Each color represents a cluster determined by the PWDA algorithm of section 4. Figure 11(left) corresponds to 4500 ALU pairwise aligned Gene Sequences with 8 clusters [37] and 11(right) to 4000 Patient Records with 8 clusters from [38]](image)

Multidimensional scaling (MDS) is a general term for a collection of techniques to configure data points with proximity information, typically dissimilarity (interpoint distance), into a target space which is normally Euclidean low-dimensional space. Formally, the $N \times N$ dissimilarity matrix $\Delta = (\delta_{ij})$ should be satisfied symmetric ($\delta_{ij} = \delta_{ji}$), nonnegative ($\delta_{ij} \geq 0$), and zero diagonal elements ($\delta_{ii} = 0$) conditions. From given dissimilarity matrix $\Delta$, a configuration of points is constructed by the MDS algorithm in a Euclidean target space with dimension $p$. The output of MDS algorithm can be an $N \times p$ configuration matrix $X$, whose rows represent each data point $x_i$ in Euclidean $p$-dimensional space. From configuration matrix $X$, it is easy to compute the Euclidean interpoint distance $d_{ij}(X) = ||x_i - x_j||$ among $N$ configured points in the target space and
to build the $N \times N$ Euclidean interpoint distance matrix $D(X) = (d_{ij}(X))$. The purpose of MDS algorithm is to map the given points into the target $p$-dimensional space, while the interpoint distance $d_{ij}(X)$ is approximated to $\delta_{ij}$ with different MDS forms correspondingly to different measures of the discrepancy between $d_{ij}(X)$ and $\delta_{ij}$. STRESS [39] and SSTRESS [40] were suggested as objective functions of MDS algorithms. STRESS ($\sigma$ or $\sigma(X)$) criterion (Eq. (3.1)) is a weighted squared error between distance of configured points and corresponding dissimilarity, but SSTRESS ($\sigma^2$ or $\sigma^2(X)$) criterion (Eq. (3.2)) is a weighted squared error between squared distance of configured points and corresponding squared dissimilarity.

$$\sigma(X) = \sum_{i<j\leq n} w_{ij} (d_{ij}(X) - \delta_{ij})^2 \quad (3.1)$$

$$\sigma^2(X) = \sum_{i<j\leq n} w_{ij} [(d_{ij}(X))^2 - (\delta_{ij})^2]^2 \quad (3.2)$$

where $w_{ij}$ is a weight value, so $w_{ij} \geq 0$. Therefore, the MDS can be thought of as an optimization problem, which is minimization of the STRESS or SSTRESS criteria during constructing a configuration of points in the $p$-dimension target space.

3.1. Scaling by MAjorizing a COmplicated Function (SMACOF)

Scaling by MAjorizing a COmplicated Function (SMACOF) [33-34] is an iterative majorization algorithm in order to minimize objective function of MDS. SMACOF is likely to find a local not global minima as is well known from gradient descent methods. Nevertheless, it is powerful since it guarantees a monotonic decrease of the objective function. The procedure of SMACOF is described in Algorithm 1. For the mathematical details of SMACOF, please refer to [32].

**Algorithm 1 SMACOF algorithm**

$Z \leftarrow X^{[0]}$;
$k \leftarrow 0$;
$\varepsilon \leftarrow$ small positive number;
$MAX \leftarrow$ maximum iteration;
Compute $\sigma^{[0]} = \sigma(X^{[0]})$;

while $k = 0$ or $(\Delta \sigma(X^{[k]}) > \varepsilon$ and $k < MAX$ do

$k \leftarrow k + 1$;
$X^{[k]} = V^T \Lambda(X^{[k-1]})X^{[k-1]}$
Compute $\sigma^{[k]} = \sigma(X^{[k]})$
$Z \leftarrow X^{[k]}$;
end while
return $Z$;

3.2. Distributed-Memory Parallel SMACOF

In order to implement distributed-memory parallel SMACOF, one must address two issues: one is the data decomposition where we choose block matrix decomposition for our SMACOF implementation since it involves matrix multiplication iterated over
successive gradient descents, and the other is the required communication between decomposed processes. For the data decomposition, our implementation allows users to choose the number of row-blocks and column-blocks with a constraint that the product of the number of row-blocks and column-blocks should be equal to the number of processes, so that each process will be assigned corresponding decomposed sub-matrix. For instance, if we run this program with 16 processes, then users can decompose the $N \times N$ full matrices into not only $4 \times 4$ block matrices but also $16 \times 1$, $8 \times 2$, $2 \times 8$, and $1 \times 16$ block matrices. In addition, message passing interface (MPI) is used to communicate between processes, and MPI.NET is used for the communication.

3.2.1. Advantages of Distributed-memory Parallel SMACOF

The running time of SMACOF algorithm is $O(N^2)$. Though matrix multiplication of $V^\top \cdot B(X)$ takes $O(N^3)$, you can reduce the computation time by using associativity of matrix multiplication. By the associative property of the matrix multiplication, $(V^\top \cdot B(X)) \cdot X$ is equal to $V^\top \cdot (B(X) \cdot X)$. While the former takes the order of $O(N^3 N^2 p)$, the latter takes only $O(2N^2 N^2 p)$, where $N$ is the number of points and $p$ is the target dimension that we would like to find a configuration for given data. Normally, the target dimension $p$ is two or three for the visualization, so $p$ could be considered as a constant for computational complexity. Also, SMACOF algorithm uses at least four full $N \times N$ double matrices, i.e. $A$, $D$, $V^\top$, and $B(X)$, which means at least $32 \times N^2$ bytes of memory should be allocated to run SMACOF program.

As in general, there are temporal and spatial advantages when we use distributed-memory parallelism. First, computational advantage should be achieved by both shared-memory and distributed-memory parallel implementation of SMACOF. While shared-memory parallelism is limited by the number of processors (or cores) in a single machine, distributed-memory parallelism can be extended the available number of processors (or cores) as much as machines are available, theoretically. SMACOF algorithm uses at least $32 \times N^2$ bytes of memory as we mentioned above. For example, 32MB, 3.2GB, 12.8GB, and 320GB are necessary for $N = 1000$, $10000$, $20000$, $100000$, correspondingly. Therefore, a multicore workstation, which has a 8GB of memory will be able to run SMACOF algorithm with 10000 data points. However, this workstation cannot be used to run the same algorithm with 20000 data points. Shared memory parallelism increases performance but does not increase size of problem that can be addressed. Thus, the distributed-memory parallelism allows us to run SMACOF algorithm with much more data, and this benefit is quite important in the era of a data deluge.

3.3. Experimental Results and Analysis

For the performance experiments of the distributed-memory parallel SMACOF, we use two nodes of Ref C and one node of Ref D in Table 1. For the performance test, we generate artificial random data set which is in 8-centered Gaussian distribution in 4-dimension with different number of data points, such as 128, 256, 512, 1024, 2048, and 4096.

Due to gradient descent attribute of SMACOF algorithm, the final solution highly depends on the initial mapping. Thus, it is appropriate to use random initial mapping for the SMACOF algorithm unless specific prior initial mapping exists, and to run several times to increase the probability to get better solution. If the initial mapping is
different, however, the computation amount can be varied whenever the application runs, so that we could not measure any performance comparison between two experimental setups, since it could be inconsistent. Therefore, the random seed is fixed for the performance measures of this paper to generate the same answer and the same necessary computation for the same problem. The stop condition threshold value (ε) is also fixed for each data. We will investigate the dependence on starting point more thoroughly using other approaches discussed in section 3.4.

3.3.1. Performance Analysis

For the purpose of performance comparison, we implemented the sequential version of SMACOF algorithm. The sequential SMACOF is executed on each test node, and the test results are in Table 3. Note that the running time of D is almost twice faster than the other two nodes, though the core’s clock speed of each node is similar. The reason would be the cache memory size. L2 cache of two Ref C nodes (C1 and C2) is much smaller than that of D node.

Table 3. Sequential Running time in seconds on each test node

<table>
<thead>
<tr>
<th>Data size</th>
<th>C1</th>
<th>C2</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.3437</td>
<td>0.3344</td>
<td>0.1685</td>
</tr>
<tr>
<td>256</td>
<td>1.9031</td>
<td>1.9156</td>
<td>0.9204</td>
</tr>
<tr>
<td>512</td>
<td>9.128</td>
<td>9.2312</td>
<td>4.8456</td>
</tr>
<tr>
<td>1024</td>
<td>32.2871</td>
<td>32.356</td>
<td>18.1281</td>
</tr>
<tr>
<td>2048</td>
<td>150.5793</td>
<td>150.949</td>
<td>83.4924</td>
</tr>
<tr>
<td>4096</td>
<td>722.3845</td>
<td>722.9172</td>
<td>384.7344</td>
</tr>
</tbody>
</table>

Initially we measured the performance of the distributed-memory parallel SMACOF (MPI_SMACOF) on each test node only. Figure 12 shows the speedup of each test node with different number of processes. Both axes of the Figure 12 are in logarithmic scale. As the Figure 12 depicted, the MPI_SMACOF is not good for small data, such as 128 and 256 data points. However, for larger data, i.e. 512 and more data points, the MPI_SMACOF shows great performance on the test data. You should notice those speedup values of larger data, such as 1024 or more data points on C1 and C2 nodes are bigger than the actual processes number using the MPI_SMACOF application, which corresponds to super-linear speedup. However, on the D node, it represented good speedup but not super-linear speedup at all. The reason of super-linear speedup is related to cache-hit ratio, as we discussed about sequential running results. MPI_SMACOF implemented in the way of block decomposition, so that those sub-matrix would be better matched in the cache line size and the portion of sub-matrix which is in cache memory at a moment would be bigger than the portion of whole matrix in it. The Figure 12 also describes that the speedup ratio (or efficiency) becomes worse when you run MPI_SMACOF with more processes on single node. It seems natural that as the number of computing units increases, the assigned computing job will be decreased but the communication overhead will be increased.
In addition, we have measured the performance of the proposed MPI_SMACOF algorithm on all the three test nodes with different number of processes. Figure 13 illustrates the speedup of those experiments with respect to the average of the sequential SMACOF running time on each node. The comparison with average might be reasonable since, for every test case, the processes are equally spread as much as possible on those three test nodes except the case of 56 processes running. The Figure 13 represents that the speedup values are increasing as the data size is getting bigger. This result shows that the communication overhead on different nodes is larger than communication overhead on single node, so that the speedup is still increasing, even with large test data such as 2048 and 4096 points, instead of being converged as in Figure 12.
3.4. Conclusions

We have developed a dimension mapping tool that is broadly applicable as it only uses dissimilarity values and does not require the points to be in a vector space. We have good parallel performance and are starting to use it for science applications as illustrated in figure 11. In later work, we will compare the method described with alternatives that can also be parallelized and avoid the steepest descent approach of SMACOF which can lead to local minima. One approach, first described in [41] and [42], uses deterministic annealing based on ideas sketched in section 4. This still uses Expectation Maximization (EM) (steepest descent) but only for the small steps needed as temperature is decreased. We will also implement the straightforward but possibly best method from ref [43] that solves equations (3.1) and (3.2) as $\chi^2$ problems and uses optimal solution methods for this.

4. Multicore Clustering

4.1. Algorithms

Clustering can be viewed as an optimization problem that determines a set of K clusters by minimizing

$$H_{VECDA} = \sum_{i=1}^{N} \sum_{k=1}^{K} M_i(k) D_{VEC}(i,k)$$

(4.1)

where $D_{VEC}(i,k)$ is the distance between point $i$ and cluster center $k$. $N$ is the number of points and $M_i(k)$ is the probability that point $i$ belongs to cluster $k$. This is the vector version and one obtains the pairwise distance model with:

$$H_{PWDA} = 0.5 \sum_{i=1}^{N} \sum_{j=1}^{N} D(i,j) \sum_{k=1}^{K} M_i(k) M_j(k) / C(k)$$

(4.2)

and $C(k) = \sum_{i=1}^{N} M_i(k)$ is the expected number of points in the k’th cluster. $D(i,j)$ is pairwise distance between points 1 and $j$. Equation (4.1) requires one be able to calculate the distance between a point $i$ and the cluster center $k$ and this is only possible when one knows the vectors corresponding to the points $i$. (4.2) reduces to (4.1) when one inserts vector formulæ and drops terms that average to zero. The formulation (4.2) is important as there are many important clustering applications where one only knows distances between points and not a Euclidean vector representation.

One must minimize (4.1) or (4.2) as a function of cluster centers for case VECDA and cluster assignments $M_i(k)$ for case PWDA. One can derive deterministic annealing from an informatics theoretic [17] or physics formalism [18]. In latter case one smoothes out the cost function (4.1) or (4.2) by averaging with the Gibbs distribution $\exp(-H/T)$. This implies in a physics language that one is minimizing not $H$ but the free energy $F$ at temperature $T$ and entropy $S$

$$F = H - TS$$

(4.3)

For VECDA and Hamiltonian $H$ given by equation (4.1), one can do this averaging exactly.

$$M_i(k) = \exp\left(- \frac{D_{VEC}(i,k)}{T}\right) / Z_i$$

(4.4)

$$Z_i = \sum_k \exp\left(- \frac{D_{VEC}(i,k)}{T}\right)$$

(4.5)

$$F = - T \sum_{i=1}^{N} \log[Z_i] / N$$

(4.6)
For the case of equation (4.2) where only distances are known, the integrals with the Gibbs function are intractable analytically as the degrees of freedom $M_i(k)$ appear quadratically in the exponential. In the more familiar simulated annealing approach to optimization, these integrals are effectively performed by Monte Carlo. This implies simulated annealing is always applicable but is usually very slow. The applicability of deterministic annealing was enhanced by the important observation in [18] that one can use an approximate Hamiltonian $H_0$ and average with $\exp(-H_0/T)$. For pairwise clustering (4.2), one uses the form motivated by the VECDA formalism (4.4).

\[
H_0 = \sum_{i=1}^{N} \sum_{k=1}^{K} M_i(k) \varepsilon_i(k) \quad (4.7)
\]

\[
M_i(k) \propto \exp(-\varepsilon_i(k)/T) \text{ with } \sum_{k=1}^{K} M_i(k) = 1 \quad (4.8)
\]

$\varepsilon_i(k)$ are new degrees of freedom. This averaging removes local minima and is designed so that at high temperatures one starts with one cluster. As temperature is lowered one minimizes the Free Energy (4.3) with respective to the degrees of freedom. A critical observation of Rose [17] allows one to determine when to introduce new clusters. As in usual expectation maximization (steepest descent) the first derivative of equation (4.3) is set to zero to find new estimates for $M_i(k)$ and other parameters such as cluster centers for VECDA. Then one looks at the second derivative $\Gamma$ of $F$ to find instabilities that are resolved by splitting clusters. One does not examine the full matrix but the submatrices coming from restricting $\Gamma$ to variations of the parameters of a single cluster with the K-1 other clusters fixed and multiple identical clusters placed at location of clusters whose stability one investigates. As temperature is lowered one finds that clusters naturally split and one can easily understand this from the analytic form for $\Gamma$. The previous work [18] on PWDA was incomplete and did not consider calculation of $\Gamma$ but rather only assumed an a priori fixed number of clusters. We have completed the formalism and implemented it in parallel. Note we only need to find the single lowest eigenvalue of $\Gamma$ (restricted to varying one cluster). This is implemented as power (Arnoldi) method. One splits the cluster if its restricted $\Gamma$ has a negative eigenvalue and this is the smallest when looked at over all clusters.

**Figure 14.** Preliminary stage of clustering shown in figure 11(left) corresponding to 4500 ALU pairwise aligned Gene Sequences with 2 clusters [37]
The formalism for VECDA can be found in our earlier work and [17]. Here we just give results for the more complex PWDA and use it to illustrate both methods. We let indices $k \mu \lambda$ runs over clusters from 1 to $K$ while $i j \alpha \beta$ run over data points from 1 to $N$. $M_i(k)$ has already been given in equation (4.8). Then one calculates:

$$A(k) = -0.5 \sum_{i=1}^{N} \sum_{j=1}^{N} D(i,j) M_i(k) M_j(k) / C(k)^2$$

$$B_\alpha(k) = \sum_{i=1}^{N} D(i, \alpha) M_i(k) / C(k)$$

$$C(k) = \sum_{i=1}^{N} M_i(k)$$

Allowing one to derive the estimate $\varepsilon_\alpha(k) = (B_\alpha(k) + A(k))$ (4.10)

Equation (4.10) minimizes $F$ of equation (4.3). The $NK \times NK$ second derivative matrix $\Gamma$ is given by:

$$\Gamma_{\alpha \beta}^{\mu \lambda} = (1/T) \delta_{\alpha \beta} \{ M_\alpha(\mu) \delta_{\mu \lambda} - M_\alpha(\mu) M_\beta(\lambda) \} + (M_\alpha(\mu) M_\beta(\lambda) / T^2) \{ \sum_{k=1}^{K} [-2A(k) - B_\beta(k) - B_\alpha(k) + D(\alpha, \beta)] [M_\alpha(k) - \delta_{k \mu}] [M_\beta(k) - \delta_{k \lambda}]/C(k) \}$$

Equations (4.9) and (4.10) followed by (4.8) represent the basic steepest descent iteration (Expectation Maximization) that is performed at fixed temperature until the estimate for $\varepsilon_\alpha(k)$ is converged. Note steepest descent is a reasonable approach for deterministic annealing as one has smoothed the cost function to remove (some) local minima. Then one decides whether to split a cluster from the eigenvalues of $\Gamma$ as discussed above. If splitting is not called for, one reduces the temperature and repeats equations (4.8) through (4.11). There is an elegant method of deciding when to stop based on the fractional freezing factors $\Phi(k)$

$$\Phi(k) = \sum_{i=1}^{N} M_i(k) (1 - M_i(k)) / C(k)$$

As temperatures are lowered after final split, then the $M_i(k)$ tend to either 0 or 1 so $\Phi(k)$ tends to zero. We currently stop when all the freezing factors are < 0.002 but obviously this precise value is ad-hoc.

### 4.2. Multi-Scale and Deterministic Annealing

In references [12] and [14], we explain how a single formalism describes many different problems: VECDA (Clustering of points defined by vectors with deterministic annealing) [16-17], Gaussian Mixture Models (GMM) [44]; Gaussian Mixture Models with deterministic annealing (GMMDA) [45]; and Generative Topographic Maps (GTM) [29]. One can also add deterministic annealing to GTM and we are currently working on this for Web applications [46]. Deterministic annealing can be considered as a multi-scale approach as quantities are weighted by $\exp(-D/T)$ for distances $D$ and temperature $T$. Thus at a given temperature $T$, the algorithm is only sensitive to distances $D$ larger than or of order $T$. One starts at high temperatures (determined by largest distance scale in problem) and reduce temperature (typically by 1% each iteration) until you reach either the distance scale or number of clusters desired. As explained in original papers [16], clusters emerge as phase transitions as one lowers the temperature and need not be put in by hand. For example the eight clusters in figure 11(left) were found systematically with clusters being added as one reduced temperature so that at a higher temperature one first split from one to two clusters to
find results of figure 14. The splits are determined from the structure of second
derivative matrix equation (4.11) and figure 11(left) is for example found by continuing
to reduce the temperature from intermediate result in figure 14.

4.3. Operational Use of Clustering and MDS

The original data is clustered with VECDA (see earlier papers for examples) or PWDA
and then visualized by mapping points to 3D with MDS as described in section 3 and
visualizing with a 3D viewer written in DirectX. As a next step, we will allow users to
select regions either from clustering or MDS and drill down into the substructure in this
region. Like the simpler linear principal component analysis, MDS of a sub-region is
generally totally different from that of full space. We note here that deterministic
annealing can also be used to avoid local minima in MDS [47]. We will report our
extensions of the original approach in [41-42] and comparison with Newton’s method
for MDS [43] elsewhere.

Clustering in high dimensions d is not intuitive geometrically as the volume of a
cluster of radius R is proportional to $R^{(d+1)}$ implying that a cluster occupying 0.1\% of
total volume has a radius reduced by only a factor 0.99 from that of overall space with
d=1000 (a value typical of gene sequences). These conceptual difficulties are avoided
by the pairwise approach. One does see the original high dimension when projecting
points to 3D for visualization as they tend to appear on surface of the lower
dimensional space. This can be avoided as discussed in [42] by a mapping Distance D $ightarrow f(D)$ where $f$ is a monotonic function designed so that the transformed distances $f(D)$
are distributed uniformly in a lower $d_L$ dimensional space. We experimented with $d_L = 2$ and 4 where the mapping is analytically easy but found it did not improve the
visualization. Typical results are shown in figure 15(right) that maps data of figure
15(left) to 2 dimensions before applying MDS – the clustering is still performed on
original unmapped data. Certainly the tendency in figure 15(left) to be at edge of
visualisation volume is removed but data understanding does not seem improved. This
approach finds an effective dimension $d_{eff}$ for original data by comparing mean and
standard deviation of all the inter-point distances $D(i,j)$ with those in a dimension $d_{eff}$. This
determines an effective dimension $d_{eff}$ of 40-50 for sequence data and about 5 for
medical record data; in each case $d_{eff}$ is a dimension smaller than that of underlying
vector space. This is not surprising as any data set is a very special correlated set of
points.

Figure 15: Results of Clustering of 4500 ALU sequences into 10 clusters before (left) and after (right)
dimensional reduction described in text below.
4.4. Parallelism

The vector clustering model is suitable for low dimensional spaces such as our earlier work on census data [12] but the results of figures 11, 14 and 15 correspond to our implementation of PWDA – the pairwise distance clustering approach of [18] which starts from equation (4.2) and its structure has similarities to familiar O(N^2) problems such as (astrophysical) particle dynamics. As N is potentially of order a million we see that both MDS and pairwise clustering are potential supercomputing data analysis applications. The parallelism for clustering is straightforward data parallelism with the N points divided equally between the P parallel units. This is the basis of most MapReduce algorithms and clustering was proposed as a MapReduce application in [7]. We have in fact compared simple (K-means) clustering between versions and MapReduce and MPI in section 2 and ref. [28]. Note that VECDA should be more suitable than K-means for MapReduce as it has more computation at each iteration (MapReduce has greater overhead than MPI on communication and synchronization as shown in section 2). VECDA only uses reduction, barrier and broadcast operations in MPI and in fact MPI implementation of this algorithm is substantially simpler than the threaded version. Reduction, Barrier and Broadcast are all single statements in MPI but require several statements – especially for reduction – in the threaded case. Reduction is not difficult in threaded case but requires care with many opportunities for incorrect or inefficient implementations.

PWDA is also data parallel over points and its O(N^2) structure is tackled similarly to other O(N^2) algorithms by dividing the points between parallel units. Each MPI process also stores the distances D(i,j) for all points i for which process is responsible. Of course the threads inside this process can share all these distances stored in common memory of a multicore node. There are subtle algorithms familiar from N-body particle dynamics where a factor of 2 in storage (and in computation) is saved by using the symmetry D(i,j) = D(j,i) but this did not seem useful in this case. The MPI parallel algorithm now needs MPI_SENDRECV to exchange information about the distributed vectors; i.e. one needs to know about all components of vectors M, B, and the vector A_i iterated in finding maximal eigenvectors. This exchange of information can either be done with a broadcast or as in results reported here by send-receive in ring structure as used in O(N^2) particle dynamics problems. We measured the separate times in the four components of MPI – namely send-receive, Reduction, and Broadcast and only the first two are significant reaching 5-25% of total time with Broadcast typically less than 0.1% of execution time. The time needed for MPI send-receive is typically 2 to 3 times that for reduction but the latter is a non trivial overhead (often 5-10%). Obviously broadcast time would go up if it was used in place of send-receive in information exchange step.

4.5. Computational Complexity

The vector and pairwise clustering methods have very different and complementary computational complexities. VECDA execution time is proportional to N d^2 for N points – each of dimension d. PWDA has an execution time proportional to N^2. PWDA can rapidly become a supercomputer computation. For example with 4500 sequence data points and 8 clusters, the sequential execution time is about 15 hours on a single core of the systems used in our benchmarks. A direct clustering with PWDA of half million points (relevant even today) would thus naturally use around 5000 cores.
(100 points per core) with pure MPI parallelization. The hybrid threading-MPI parallelism could efficiently support more cores.

We note that currently some 40-70% of the computation time is used in deciding whether to split clusters in PWDA; there are probably significantly faster algorithms here. The runs of VECDA reported here correspond to a low dimension space $d = 2$ for which negligible time is spent in splitting decision. The second derivative matrices are of size $NK \times NK$ for PWDA and of size $dK \times dK$ for VECDA. These are full matrices but as power method for determining maximal eigenvalues is used the computation is proportional to to the square of the matrix dimension. For computations reported here, the annealing uses from 1000-10,000 temperature steps while each eigenvalue determination uses 10-200 iterations.

4.6. Performance

We have performed extensive performance measurements [11-14] showing the effect of cache and for Windows runtime fluctuations can be quite significant. Here we give some typical results with figure 15 showing the performance of PWDA on the single 24 core workstation (ref D of table 1). The results are expressed as an overhead using the definitions of equation (1) introduced in section 2. We compare both MPI and thread based parallelism using Microsoft’s CCR package [20-21]. As these codes are written in C#, we use MPI.NET[35-36] finding this to allow an elegant object-based extension of traditional MPI and good performance. MPI.NET is a wrapper for the production Microsoft MPI.

Figure 16 shows that although threading and MPI both get good performance, their systematics are different. For the extreme case of 24-way parallelism, the thread implementation shows an overhead that varies between 10 and 20% depending on the data set size. MPI shows a large overhead for small datasets that decreases with increasing dataset size so in fact 24-way MPI parallelism is 20% faster than the thread version on the largest 10,000 element dataset. This is due to the different sources of the overhead. For MPI the overhead is due to the communication calls which are due to reduce (20%) and send-receive (80%) and this as expected decreases (inversely proportional to dataset size) as the dataset size increases. For threads there is no memory movement overhead but rather the overhead is due to the Windows thread scheduling that leads to large fluctuations that can have severe effects on tightly synchronized parallel codes such as those in this paper as discussed in refs. [11-14]. We see some cases where the overhead is negative (super-linear speedup) which is due to better use of cache in the higher parallelism cases compared to sequential runs. This effect is seen in all our runs but differs between the AMD and Intel architectures reflecting their different cache size and architecture.

Comparing center and right datasets we see that MPI gets comparable performance on cores of a single node (center points) or when running one process per node on up to 24 nodes of the Infiniband connected cluster. In the results plotted in the figure, MPI gets better performance (smaller overhead) than threading on the largest 10,000 element Patient dataset. This reflects the large chunks of processing per MPI process. As seen in figure this is not always the case as threading outperforms MPI on the 2000 and 4000 element datasets for largest 24-way parallelism. As a dramatic example using all 768 cores of Tempest (ref I Table 1), the pattern 24X1X32 (24 threads on each of 32 nodes connected as 32 MPI processes) runs 172 times faster than the communication dominated 1X24X32 (24 internal MPI processes on each of 32 nodes).
The fluctuations in thread execution times are illustrated in figure 17 showing standard deviations from 5 to 10% on a simple kernel representative of the VECDA clustering algorithm. The identical code (translated from C# to C) shows order of magnitude lower fluctuations when run under Linux [13] with interesting systematics even in Linux case. These fluctuations can give significant parallel overheads as parallel algorithms used in VECDA and PWDA like those in most scientific algorithms requires iterative thread synchronization at the rendezvous points. Here the execution time will be the maximum over that of all the simultaneous fluctuating threads and so increase as this number increases. As described in the earlier papers we have always seen this and reported this effect to Microsoft. We found that these fluctuations were the only sizeable new form of parallel overhead compared to those well known from traditional parallel computing i.e. in addition to load imbalance and communication overhead. We did note extra overheads due to different threads interfering on a single cache line (“false sharing”) but our current software is coded to avoid this.
Figure 17. Measurements from [11, 12] showing 5 to 10% runtime fluctuations on an 8 core workstation. The results are plotted as a function of number of simultaneous threads from 1 to 8 and for three different dataset sizes.

Figure 18. Parallel Overhead for VECDA using long lived threads run on 128 core Madrid Cluster in table 1. The results achieve a given parallelism by choosing number of nodes, MPI processes per node and threads per MPI process. The number of threads increases as you move from left to right for given level of parallelism.
Figure 19. Comparison of use of short lived (solid lines) and long lived (dashed lines) threads for the Vector-based deterministic annealing VECDA. The results achieve a given parallelism by choosing number of nodes, MPI processes per node and threads per MPI process. The number of threads increases as you move from left to right for given level of parallelism.

Note that the fluctuation effect is larger in the work reported here compared to our previous papers as we are looking here at many more simultaneous threads. Note that the effect does not just reflect the number of threads per process but also the total number of threads because the threads are synchronized not just within a process but between all processes as MPI calls will synchronize all the threads in the job. Thus it is interesting to examine this effect on the full 128 core Madrid cluster as this could even be a model for performance of future much larger core individual workstations.

We note that VECDA and PWDA differ somewhat in their performance characteristics. VECDA only uses modest size reductions (dominant use), broadcast and barrier MPI operations and so has particularly fast MPI synchronization. PWDA also has MPI_SENDRECV (exchange of data between processes) which increases the MPI synchronization time. Thus VECDA shown in figures 18 and 19 tends always to have MPI at least as fast as CCR and in some cases very much faster. Figure 18 shows the parallel overhead for 44 different choices of nodes (from 1 to 8), MPI processes per node (from 1 to 16) and threads per node (from 1 to 16 divided between the MPI processes per node). The results are divided into groups corresponding to a given total parallelism. For each group, the number of threads increases as we move from left to right. For example in the 128 way parallel group, there are five entries with the leftmost
being 16 MPI processes per node on 8 nodes (a total of 128 MPI processes) and the rightmost 16 threads on each of 8 nodes (a total of 8 MPI processes). We find an incredibly efficient pure MPI version – an overhead of just 0.08 (efficiency 92%) for 128 way parallelism whereas the rightmost case of 16 threads has a 0.63 overhead (61% efficiency). All cases with 16 threads per node show a high overhead that slowly increases as the node count increases. For example the case of 16 threads on one node has an overhead of 0.51. Note that in this we use scaled speedup i.e. the problem size increases directly according to number of parallel units. This ensures that the inner execution scenarios are identical in all 44 cases reported in figure 18. We achieve scaled datasets by replicating a base point set as one can easily see that leads to same mathematical problem but with a work that increases properly as number of execution units increases.

Figure 19 looks again at the vector clustering VECDA comparing MPI versus two versions of threading. MPI is again very efficient – the 32 way parallel code with 16 MPI processes on each of two 16 core nodes has overheads (given by equation (1) and roughly 1 – efficiency) of 0.05 to 0.10. For the case of 16 threads on each of two nodes the overhead is 0.65 (short lived) to 1.25 (long lived) threads. The short lived threads are the natural implementation with threads spawned for parallel for loops. In the long lived case, the paradigm is similar to MPI with long running threads synchronizing with rendezvous semantics.

![Graph](image_url)

**Figure 20.** Parallel Overhead for PWDA runs on a 10,000 element dataset on the 128 core cluster (Ref. F in Table 1) with patterns defined in figure 16 and in label in figure itself.

Figure 20 shows results of PWDA for a 10,000 element dataset on the 128 core cluster (ref. F in Table 1). The results show threading outperforming MPI for the highly
parallel results on right whereas on left (2- to 8-way parallelism) MPI outperforms threading. That is due to MPI being affected by the communication overhead of send-receive as discussed above for the results of figure 16. The results also show effects of the cache seen in the negative overheads (corresponding to a slow 1x1x1 case). The patterns are always labeled as (threads per process)x(MPI processes per node)x(nodes). Note figures 16 and 20 study the overhead for a fixed problem whereas figures 18 and 19 look at scaled speedup with problem size increasing proportional to number of parallel units. We see that the 10,000 element dataset can run well up even up to 128-way parallelism.

5. Conclusions

This paper has addressed several issues. It has studied the performance of a variety of different programming models on data intensive problems. It has presented novel clustering and MDS algorithms which are shown to parallelize well and could become supercomputer applications for large million point problems. It has compared MPI and threading on multicore systems showing both to be effective but with different overheads. We see these complemented by the data intensive programming models including Dryad and Hadoop as well as an in house version of MapReduce. These support an “owner stores and computes” programming paradigm that will be of increasing importance.

Acknowledgements

We thank other members of the SALSA parallel computing group including Jong Choi and Yang Ruan for conversations. Our work on patient data would not have been possible with Gilbert Liu from IU Medical School and the gene sequence data was produced by Haixu Tang and Mina Rho from the IU Bioinformatics group who also answered our endless naïve questions. The IU MPI.NET group led by Andrew Lumsdaine was very helpful as we learnt how to use this. Our research was partially supported by Microsoft. We have obtained continued help on Dryad from Roger Barga and CCR from George Chrysanthakopoulos and Henrik Frystyk Nielsen. Scott Beason provided key support on visualization and set up of Clusters. The work on Particle Physics data analysis was a collaboration with Deep Web Technologies from New Mexico and Julian Bunn at Caltech.

References

http://grids.ucs.indiana.edu/ptliupages/publications/CCGridDec07-Final.pdf


[23] Discovery, Development Platform, lightweight service-oriented runtime, and a scalable and extensible platform. For details, see http://www.erlang.org/


Processing of Large-Scale Biomedical Images on a Cluster of Multicore CPUs and GPUs

Umit V. CATALYUREK a,b,2, Timothy D. R. HARTLEY a,b, Olcay SERTEL a,b, Manuel UJALDON c, Antonio RUIZ c, Joel SALTZ d and Metin GURCAN b

a Dept. of Biomedical Informatics, The Ohio State University, USA
b Dept. of Electrical and Computer Engineering, The Ohio State University, USA
c Computer Architecture Department, University of Malaga, Spain
d Center for Comprehensive Informatics, Emory University, USA

Abstract. Today’s state-of-the-art cluster supercomputers include commodity components such as multi-core CPUs and graphics processing units. Together, these hardware devices provide unprecedented levels of performance in terms of raw GFLOPS and GFLOPS/cost. High-performance computing applications are always in search of lower execution times, greater system utilization, and better efficiency, which means that developers will need to leverage these disruptive technologies in order to take advantage of modern cluster computers’ full potential processing power. New application models and middleware systems are needed to ease the developer’s task of writing programs which efficiently use this processing capability. Here, we present the implementation of a biomedical image analysis application which serves as a case-study for the development of applications for modern heterogeneous supercomputers. We present detailed application-specific optimizations which we generalize and combine with new programming models into a blueprint for future application development. Our techniques show good success executing on a modern heterogeneous GPU cluster providing 10 TFLOPS of peak processing capability.

Introduction

High-performance computing is currently undergoing a period of rapid change. Disruptive technologies such as multi-core processors and graphics processing units (GPUs) offer excellent performance/cost ratios. As such, these technologies will quickly become a major part of modern supercomputers. Developers must learn to program these processors to take full advantage of the resources of this new breed of heterogeneous super-

1This work was partially supported by the US National Science Foundation (CNS-0643969, CNS-0403342, CNS-0615155, CFF-0342615), by the NIH NIBIB BISTI (P20EB000591), by US Department of Energy (ED-FC02-06ER25775), by the Ministry of Education of Spain (TIC2003-06623, PR-2007-0014), by the Junta de Andalucía of Spain (P06-TIC-02109), by Fundación Caixa-Castellón/Bancaixa and the University Jaume I (P1-1B2007-32), and by the young investigator award from the Children’s Neuroblastoma Cancer Foundation.

2Corresponding Author: Umit V. Catalyurek, 3190 Graves Hall, 333 W. 10th Ave., Columbus, OH 43210, USA; E-mail: catalyurek.1@osu.edu
computer. This paper represents a case study in the use of these disruptive architectures to accelerate a real-world, large-scale biomedical image analysis application. As such, it provides insight into techniques used to wrench the full performance out of the resources for a large class of important scientific applications.

Multi-core CPUs offer unprecedented amounts of processing power compared with single-core CPUs in terms of form-factor, cost, and power dissipation. While the programming methods must be adjusted to compensate for multi-cores’ different characteristics compared to those of traditional symmetric multiprocessors (SMP), it is the hybridization of low-cost cluster architectures which affects the largest groups of developers and applications. Whereas hybrid clusters of SMPs where once the realm only of costly supercomputers, multi-core CPUs turn nodes with this added complexity into a commodity.

DataCutter [1,2] is a powerful middleware tool for the development of large-scale, distributed, data-parallel applications. DataCutter handles the inherent hybrid nature of clusters of multi-core nodes, as well as the heterogeneous nature of each node with the addition of a GPU. DataCutter allows simple application decomposition, transparent task replication, and task graph execution, resulting in applications which scale well across different cluster configurations. The use of DataCutter for developing programs for modern heterogeneous, hierarchical supercomputer architectures allows the developer to focus on the application itself, rather than managing the hardware.

The newest versions of programmable GPUs provide a compelling alternative to traditional CPUs, since they offer extremely high floating point performance for scientific applications which fit their architectural idiosyncrasies [14]. This fact has attracted many researchers and encouraged the use of GPUs in many fields [8] including data mining [9], image segmentation and clustering [11], numerical methods for finite element computations used in 3D interactive simulations [26], and nuclear, gas dispersion and heat shimmering simulations [28]. In other work [19,20], by implementing an application on NVIDIA GPUs using their Cg shader language, we have achieved 45-factor speed-up versus the algorithm implemented in C++ running on a single CPU.

Both NVIDIA and AMD have released software components, which when combined with recent GPUs, provide simpler access to GPU computing power than that realized by treating the GPU as a traditional graphics processor. CUDA (Compute Unified Device Architecture) [5] is NVIDIA’s solution which provides a simple block-based API for programming NVIDIA GPUs. AMD’s solution is called AMD Stream Computing and includes such technologies such as the Brook+ [3] compiler and the Compute Abstraction Layer, both of which allow the developer to work in a high-level language which abstracts away GPUs’ specifics. Both companies also have hardware products aimed specifically at the scientific General Purpose GPU computing (GPGPU) market; the Tesla [24] products are from NVIDIA, and Firestream [7] is AMD’s product line.

For this case study of the use of disruptive architectures to improve the performance of a large-scale biomedical image analysis application, we have chosen to use NVIDIA’s CUDA to program the GPU. CUDA provides easier access to the high computational performance available in modern GPUs. Additionally, CUDA provides capabilities beyond that of other programming methods with respect to applications which do not entirely fit into the more traditional graphics processing paradigm. The use of the capabilities exposed by CUDA will prove instrumental in improving the performance of certain portions of our application.
Our goal is to develop best practices for the development of large-scale, distributed biomedical image analysis applications on a cooperative cluster of GPUs and multi-core CPUs. The advent of multi-core CPUs means that more computation can occur within a single node than was previously possible; traditional SMPs are now becoming hybridized, with multiple multi-core CPUs per node. Additionally, new architectures can support more than one GPU in a single node. Therefore, our target hardware architecture for this case study is a cluster of compute nodes with multiple multi-core CPUs and multiple GPUs.

Biomedical applications such as the one presented here are becoming a major focus due to the possible benefit to the public welfare as well as to the scientific community. In particular, imaging applications represent an opportunity for innovation - they are challenging for several reasons. In using disruptive architectures to help doctors perform their work more effectively, biomedical image analysis applications provide many opportunities for novelty.

The analysis of histopathology images is particularly challenging due to the large size of the data. Since uncompressed image sizes can be 30 gigabytes for one slice of tissue, typical datasets for case studies can easily stretch to terabytes. The computation required to analyze these images can be extensive; analysis of a single image can often take hours on a single CPU. Furthermore, in practice a biopsy sample taken from a patient usually generates 5-6 slides that are required to be analyzed.

In order to benefit a specific group of dedicated doctors, as well as to investigate techniques for improving the performance of large-scale biomedical image analysis applications by using disruptive architectures, this paper presents the implementation of a specific biomedical application on a heterogeneous cluster of multi-core CPUs and GPUs.

The rest of the paper is organized as follows. Section 1 presents the image analysis application which provides the testbed for this new paradigm of cooperation between multi-core CPUs and GPUs. Section 2 focuses on the specifics of the GPU programming with CUDA and that of the parallelization strategies. The experimental results are presented in section 3, and section 4 concludes.

1. Histopathological Image Analysis

We considered neuroblastoma (NB) as a case study of a large scale histopathological image analysis application. Mainly affecting children, NB is an aggressive cancer that develops from the nervous system. The prognosis of the disease largely depends on histopathological examination of haematoxylin and eosin (H&E)-stained tissue slides, performed visually under a microscope by expert pathologists. The current classification system is based on several morphological features of the tissue histology such as the degree of Schwannian stromal development, the grade of differentiation and the mitosis and karryorhexis index. This classification scheme known as the International Neuroblastoma Classification System (INCS) is developed by Shimada et al. and adopted as the standard prognostic system by International Neuroblastoma Pathology Committee [22]. Together with other indicators, i.e. the disease stage, genetic factors and the age of the patient, the prognosis is classified as either favorable or unfavorable. The subsequent treatment planning is based on the classification result. However, the qualitative exami-
nation of tissue samples is often subject to considerable inter- and intra-reader variations and sampling bias; it is therefore prone to error and may cause inaccurate treatment planning for the patients. To address this drawback, we are developing a computerized image analysis system for NB [4,10,19,20,21]. Quantitative image analysis is conducive to extracting more objective and accurate diagnostic clues that might not be easily observed by qualitative analysis performed by pathologists.

**Figure 1.** Sample image tile of $512 \times 512$ resolution captured at $20 \times$ exhibiting stroma-rich (upper right region) and stroma-poor (lower left region) tissue types.

In this study, we focused on the classification of the degree Schwannian stroma development, one of the prominent indicators of malignancy as defined in the INCS, which requires the classification of tissue as either stroma-rich or stroma-poor. We formulated our approach as a statistical pattern classification problem and proposed a solution using the texture characteristics of the tissue. Figure 1, shows a small image region of $512 \times 512$ resolution captured at $20 \times$, exhibiting typical stroma-rich and stroma-poor tissue regions. As can be seen in Figure 1, the texture of the stroma septa (upper right region) is quite different than the neurophil (lower left region). The hair-like fibrin structures of stroma exhibit patterns that are locally organized in particular directions, whereas the neurophil exhibits a more mesh like structure. This information has been captured by extracting texture features using co-occurrence statistics and local binary patterns (LBPs). A more detailed description of the proposed image analysis approach for the classification of stromal development can be found in [21].

Due to their very large resolutions (e.g., up to more than $100K \times 100K$ and more than 30 gigabytes), the whole-slide images are decomposed into smaller non-overlapping image tiles and the proposed image analysis routine is applied to each image tile independently. Then, by stitching together the computerized decision for each image tile, we obtain the corresponding classification map, which consists of the classification labels for each tile (stroma-rich or stroma-poor for this particular case). The granularity (i.e., the resolution) of the classification map is related to selected image tile resolution, i.e., lower resolution image tiles correspond to higher resolution classification maps. Section 1.1 gives a summary of the proposed image analysis approach that is applied to each image tile.

1.1. The algorithm

As mentioned earlier, our goal is to quantitatively differentiate the texture of stroma-rich tissue from the texture of stroma-poor tissue. Therefore, we constructed a feature vector
consisting of co-occurrence statistics and LBPs for each image tile. The classification (i.e., as either stroma-poor or stroma-rich) was performed in this feature space in a supervised manner. Figure 2 shows the flowchart of the image analysis algorithm for the classification of stromal development in H&E-stained digitized NB images [19,20]. The image analysis occurs in four main stages, discussed below.

**Color conversion.** To better represent the intensity and color information independently, we used the La*b* color space that provides a perceptually uniform color space. Perceptual uniformity means that the same amount of change in color value produces the same amount of perceptual difference of visual importance [15]. In the La*b* color space, the L channel corresponds to illumination and the a* and b* channels correspond to the two color-opponent dimensions (color opponents are colors which are perceived by the human eye as opposites). We aimed to separate color and illumination information so that the computation of texture (i.e. the spatial pattern of pixels in terms of intensity and color) would be more appropriate.

**Co-occurrence features.** Using co-occurrence statistics captured using co-occurrence matrices, four features are extracted from each channel: contrast, correlation, homogeneity and energy. These features are widely known as Haralick features [25]. Co-occurrence matrices represent how often a pixel with the intensity value \( i \) occurs in a specific spatial relationship to a pixel with the intensity value \( j \) (see Figure 3). The size of the co-occurrence matrix has a major impact on the workload, but a relatively marginal influence on the algorithm’s classification accuracy. This has been validated after comparing the classification accuracies over our training dataset by using co-occurrence matrix sizes varying between 4-64 [20]. Therefore, in our experiments we reported our timing comparisons using a \( 4 \times 4 \) co-occurrence matrix size, which yields the fastest execution times. This small co-occurrence matrix size is appropriate for this application because we are extracting complementary information from three channels, illumination (L), and opponent color channels (a* and b*). Additionally, H&E-stained tissue images have consistent high quality and high contrast, especially in color opponent a* and b* channels. Since the structures in the images are meant to be as discernible as possible, we have leveraged this high contrast to reduce the co-occurrence matrix size, and therefore the execution time of the application.

**LBP operator.** Capturing the presence of the micro-patterns in the image, the local binary pattern (LBP) is a powerful texture feature construction method widely used in applications ranging from facial expression recognition [16] to content-based image retrieval [23]. The conventional LBP operator is applied to every pixel by examining the eight neighbors to see if their intensity is greater than that of the center pixel. The results
form a binary number $b_0b_1b_2b_3b_4b_5b_6b_7$ where $b_i = 0$ if the intensity of the $i$th neighbor is less than or equal to $p$ and 1 otherwise (see Figure 4).

Ojala et al. introduced rotation-invariant LBP features by using circularly sampled neighborhood pixels [13]. Instead of using a rectangular kernel, they propose the use of a circular kernel around each pixel to increase the degree of freedom for rotational invariance. Furthermore, they introduced uniform patterns that contain, at most, two bitwise transitions from 0 to 1 or vice versa in its circular string. These uniform patterns are referred to as bright spot, flat area, dark spot, and various edges of positive and negative curvature. Following this powerful representation that is invariant to rotation and any local or global intensity change, we computed the LBP features using a radius of one pixel to construct the circular pattern with eight samples around each pixel.

Histogram Operations. Since LBP feature values are in the range $0 \rightarrow 255$, a histogram is constructed with 256 bins for the entire image, with each bin accumulating the number of pixels which have that LBP feature value. Following the uniform LBP approach in [13], these bins are then reduced into ten canonical classes and normalized between 0 and 1 to become the components of a ten-dimensional vector. The Bhattacharyya distance [18] is then measured between the LBP feature vector and a $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$ vector to constitute the uniform LBP feature value used for the stroma classification.

The subsequent classifier, which operates on the full set of features calculated for an image by the methods discussed here, is a computationally inexpensive process, so it is not considered in this study.

1.2. Major challenges

Input tissue samples are digitized at 40x magnification and stored in TIFF files using JPEG compression and the RGB color format. Each whole-slide image has a resolution of up to $110K \times 80K$ pixels in the worst case, which poses two major challenges:
The algorithm takes close to an hour to run on a single CPU for even a small image (see Figure 11). This motivates us to look for alternative methods to accelerate the computation. Our target hardware architecture is therefore a cluster of hybrid, multi-core, multi-socket nodes with GPUs distributed among all of the nodes.

A single uncompressed image’s size is well over the memory capacity of a single GPU. Additionally, balancing the load effectively across the whole computational cluster requires a smaller data granularity. In order to enable efficient GPU computation and to allow for effective parallelization of the computation across multiple nodes, we split the image into 1K x 1K tiles.

Beyond its potential for improving the diagnostic capability for expert pathologists, our biomedical image analysis application provides an opportunity to develop novel methods for general purpose computing on GPUs, because it exhibits a diverse set of characteristics. For example, color conversion is a typical streaming operation with no data reuse. Co-occurrence feature construction, on the other hand, exhibits a large degree of data reuse and locality, which typically favors a deep cache hierarchy such as is found in modern CPUs. The LBP operator and histogram construction both use extensive indirect array accesses such as those characterizing irregular computing and reduction operations often found in linear algebra. These two phases present undesirable features for both the CPU and the GPU. Therefore, we use this case-study application to investigate the techniques required for a cluster of GPU-equipped nodes to fully exploit each architecture to achieve high performance.

2. Heterogeneous GPU cluster programming with CUDA and DataCutter

In order to illustrate the burgeoning cooperative relationship between GPUs and multi-core CPUs, this section shows the details of the CUDA [5] implementation of the image analysis algorithm as well as the filter-stream parallelization using DataCutter middleware [1,2]. The programming tools and paradigms used in this work are summarized in Figure 5.

Since our biomedical image analysis applications has its roots in a sequential Matlab implementation (as many domain-specific image analysis algorithms do), the initial implementation of the algorithm is this version. The most natural step for developers to take once more performance is required would be to implement the algorithm in a lower-level language such as C or C++. A large performance increase can be realized from this re-implementation, due to Matlab being an interpreted, very high-level language.

Before the increased popularity of GPGPU programming frameworks, algorithms needed to be implemented as if they were graphics applications. As such, the first GPU implementation of our image analysis was in NVIDIA’s C-like shader programming language, Cg. This work was presented in [20]. The next version of the GPU image analysis algorithm is presented in section 2.1, and represents a large step forwards in terms of execution time reductions.

In order to take full advantage of even a single node of a heterogeneous GPU cluster equipped with multi-core CPUs, some type of shared-memory programming method must be employed. Standard POSIX threads is one option, but we make the case here for a more powerful streaming programming method and runtime engine called DataCutter. While a message-passing library such as MPI would allow a developer to take advantage
of more than one node in a GPU cluster, we believe DataCutter to be a superior solution for this type of application. We make the case for its superiority in section 2.2.

2.1. CUDA

The Compute Unified Device Architecture (CUDA) is a programming interface and set of supported hardware to enable general purpose computation on NVIDIA GPUs. The programming interface is ANSI C with some specific additional keywords, constructs, and library functions. A special compiler generates the executable code for the GPU. Since CUDA is specifically designed for general-purpose computing, it can leverage special hardware features not available to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight thread context switching.

NVIDIA’s G80 core was the first device to support CUDA, which was introduced in 2007. The hardware product we use in our experiments - the 8800 GTX - has a peak multiply-add single-precision performance of 330 GFLOPS, features 76.8 GB/s memory bandwidth (mainly due to a 384-bit wide bus), contains 1.5 GB of video memory split into six partitions of 256 MB (each 64 bits wide), and incurs negligible penalties when creating thousands of threads.

As a parallel architecture, the G80 becomes a SIMD (Single Instruction Multiple Data) processor endowed with 128 cores. Cores are organized into 16 multi-processors, each having a set of 8,192 32-bit registers, a 16 KB shared memory which acts like a software-controlled data cache close to the registers in access speed, and constants and texture caches of a few kilobytes. Each multi-processor can keep track of a variable number of thread contexts; local resources are divided among them. In any given cycle, each core in a multi-processor executes the same instruction on different data, and communication between multi-processors is performed through global memory (see Figure 6).
Figure 6. The CUDA hardware interface for the G80 core.

NVIDIA has committed themselves to CUDA such that future architectures (such as their recently released GT200 architecture) will support the same executables. While future architectures may have more cores or more registers per multi-processor, some of the CUDA restrictions will stay in place. The CUDA SDK Readme files detail what the restrictions are for each CUDA-supported architecture.

2.1.1. Programming elements

There are some important elements involved in the conception of a CUDA program which are key for understanding the programming model as well as the optimizations we have performed during the implementation. We describe them below and Figure 7 shows a sample block diagram.

- A program is decomposed into **blocks** that run *logically* in parallel (physically only if there are resources available). Assembled by the developer, a block is a group of threads that is mapped to a single multi-processor, where they can share 16 KB of memory (see Figure 6). All the threads in blocks concurrently assigned to a single multi-processor divide the multi-processor's resources equally amongst themselves. The data is also divided amongst all of the threads in SIMD fashion explicitly managed by the developer.

- A **warp** is a collection of threads that can physically run concurrently on all of the multi-processors. The size of the warp (32 threads on the G80) is less than the total number of cores (128 on the G80, 240 on the newer GT200 architecture) due to memory access limitations. The developer has the freedom to determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, they are time-shared on the actual hardware resources. This can be advantageous, since time-sharing the ALU resources amongst multiple threads can overlap the memory latencies when fetching ALU operands.
• A **kernel** is the actual code to be executed by the thousands of threads in a CUDA program, which share the same executable and global address space. Threads do not need to follow exactly the same path of execution, since conditional execution of different operations within each multi-processor can be achieved based on a unique thread ID.

• A **grid** is a collection of all blocks in a single CUDA executable. Grid sizes and dimensions are explicitly defined by the application developer during the function call; these grid characteristics affect the way blocks of threads are scheduled by the runtime system. This in turn affects the way the hardware groups threads in warps, affecting performance. Thus, the grid layout is important and deserves the developer’s attention.

At the highest level, a program is decomposed into kernels. Each kernel is scheduled to run on the hardware by a grid composed of blocks of threads. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided except for the fact that memory stays consistent from one grid-kernel execution to the next.

### 2.1.2. Optimizations

Major hardware and software limitations are listed in Table 1, where they are ranked according to their influence on code development and performance based on our experience. Managing those limits is critical when optimizing applications. Developers still have a great deal of freedom in determining CUDA program layouts, though often unintended side-effects occur when testing strategies to avoid one limit, causing other limits to be hit.

We present two techniques which are useful to employ when optimizing a CUDA application.
Table 1. Major hardware and software limitations with CUDA. Constraints are listed for the G80 GPU and categorized according to their difficulty of optimization and impact on the overall performance.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Limitation</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Processors per GPU</td>
<td>16</td>
<td>Low</td>
</tr>
<tr>
<td>Processors / Multi-Processor</td>
<td>8</td>
<td>Low</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>Low</td>
</tr>
<tr>
<td>Thread Blocks / Multi-Processor</td>
<td>8</td>
<td>Medium</td>
</tr>
<tr>
<td>Threads / Block</td>
<td>512</td>
<td>Medium</td>
</tr>
<tr>
<td>Threads / Multi-Processor</td>
<td>768</td>
<td>High</td>
</tr>
<tr>
<td>32-bit registers / Multi-Processor</td>
<td>8192</td>
<td>High</td>
</tr>
<tr>
<td>Shared Memory / Multi-Processor</td>
<td>16 Kbytes</td>
<td>High</td>
</tr>
</tbody>
</table>

1. **Ensure shared memory is used wisely.** As a software-controlled data cache, using the shared memory region inside a multi-processor is critical for high application performance. Even though global memory delivers impressive bandwidth, these memory accesses incur a large latency penalty for fetching arithmetic operands (around 400 times slower than shared memory). Complex kernels are often register-bound, meaning that they cannot give sufficient thread parallelism to fully hide this global-memory access latency. This leads to the second technique.

2. **Organize threads in blocks to maximize parallelism, enhance hardware occupancy (the number of threads in a warp which can execute simultaneously) and avoid memory banks conflicts.** In this respect, the CUDA documentation provides a hint, where NVIDIA suggests a single block contain between 128 and 256 threads to maximize execution efficiency. A tool developed by NVIDIA, the CUDA Occupancy Calculator, may also be used as a guide to attain this goal.

We address each of these issues separately now.

**Memory usage.** Attention must be paid to how the threads access the 16 banks of shared memory, since only when the data accessed resides in different banks can all of the available ALU bandwidth truly be used.

Each bank only supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multi-processor’s running threads until their operands arrive. The use of up to 16 KB of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely. However, although the effort applied can be very rewarding, this type of optimization is often very difficult. Execution times may decrease by as much as 10x for vector operations and latency hiding may increase by up to 2.5x [6].

**Thread organization.** When developing applications for GPUs with CUDA, the management of registers becomes important as a limiting factor for the amount of parallelism we can exploit. Each multi-processor contains 8,192 registers which will be split evenly among all the threads of the blocks assigned to that multi-processor. Hence, the number
of registers needed in the computation will affect the number of threads which can be executed simultaneously, given by the constraints outlined in Table 1.

For example, when a kernel instance consumes 16 registers, only 512 threads can be assigned to a single multi-processor. This can be achieved by using one block with 512 threads, two blocks of 256 threads, and so on. By this point, it is clear that while there are some tools to help with determining CUDA application layouts, some iterative optimization is required to achieve the lowest execution time.

The thread deployment chosen was different for each of our biomedical image analysis algorithm’s kernels, so we describe this implementation detail for each stage. Below, we refer to each of the four computationally demanding algorithm stages (see Table 2) by its order of execution in the CUDA implementation. Phase 1 is the Color conversion, phase 2 is the Co-occurrence feature calculation, phase 3 is the LBP operator, and phase 4 is the histogram calculation.

2.1.3. Implementation Details of Image Analysis Application

In order to achieve the best application execution time, we have used what we consider to be a typical CUDA development cycle, which we will describe briefly. First, the initial implementation was compiled using the CUDA compiler and a special -cubin flag that outputs the hardware resources (memory and registers) consumed by the kernel. Using these values in conjunction with the CUDA Occupancy Calculator, we were able to analytically determine the number of threads and blocks that were needed to use a multi-processor with maximum efficiency. When enough efficiency was not achieved, the code was revised to reduce the register requirement.

Due to the high floating point computation performance of the GPU, memory access becomes the bottleneck in many parts of our application. Even though during each phase of the algorithm, we only operate on 1K×1K tiles instead of the entire multi-gigabyte image, the input image tile’s data size (1K×1K×3 bytes) is much larger than the size of the multi-processor shared memory (16 KB), so we prioritize data structures like partial co-occurrence matrices (used in phase 2) and partial histograms (phase 4). In phase 1, even though the tile pixels are read once without being reused, the execution time was lower when using shared memory (2.32 ms versus 2.77 ms - see Table 2). Also, in phase 3, the input pixels were moved to shared memory because the calculation of the LBP feature shows high data reuse.

In order to illustrate the progression of a typical CUDA implementation, we now discuss the specific optimizations applied to each phase of the image analysis application. Table 2 shows the result of the major optimizations on the execution time of each of the algorithm stages.

**Phase 1: Color conversion** As a baseline implementation, we initially used 24 bit float3 data types for each color channel. However, extra performance was obtained by padding the RGB input to 32 bits, which simplifies all subsequent optimizations involving shared memory. This is called data coalescing, and for this phase it saved 35% of the computation time at the expense of communication time (see Table 2). Next, it was found that 8 bit uchar data types were sufficient for each R, G, B color channel for the precision of the application. As expected, this reduces the communication time by nearly a factor of 4.

We then used the special CUDA compilation flag to find that the color conversion kernel requires 13 registers and 1,064 bytes of shared memory, leading to a maximum
Figure 8. Phase 2: Local co-occurrence matrices in CUDA. Assigning banks in shared memory to each thread to avoid conflicts when computing co-occurrence matrices.

Processor occupancy of 75% when allocating between 176 and 192 threads. However, we chose to allocate 256 threads instead, trading processor occupancy (from 75% down to 67%) for better load balance, since 256 is a multiple of 32 (maximum threads per warp) and a divisor of 768 (maximum threads per multi-processor) and 1024 (maximum pixels per image). The result was that the execution times improved slightly (see the highlighted version in Table 2 versus the last version outline, which reports the minimum time obtained for all threads/block cases between 169 and 192, which turns out to be 169 threads). Unfortunately, maximum performance here is limited because each thread needs more than 10 registers (11 to be precise), which, as discussed earlier, prevents us from reaching the maximum occupancy of 768 threads within a multi-processor. The optimal execution time for this phase was 2.98 ms for pixel transfer and 2.32 ms for computing the color conversion, as reflected in Table 2.
Phase 3: LBP operator in CUDA. We allocate more memory than computing threads and overlap two rows and columns of the external frame on neighbor blocks in order to cover the whole computational domain homogeneously.

**Phase 2: Co-occurrence features** This kernel requires 9 registers and 4132 bytes of shared memory, which allowed us to allocate 3 parallel blocks of 256 threads. This perfect usage of all 768 threads filled all of the G80 hardware resources for a 100% occupancy factor. Pixels are equally distributed among threads and local co-occurrence matrices are simultaneously computed within them. Finally, partial results are accumulated through a reduction operator.

It was found to be challenging to compute co-occurrence matrices in shared memory while avoiding conflicts accessing its 16 banks. With a grid of 256 threads arranged in a 16x16 grid, the naive thread deployment would force the 32 threads in a warp to access only 8 shared-memory banks, which would severely limit parallelism and performance. We found that by intelligently shuffling the active threads combined into a warp, the local matrix computation and the subsequent global matrix aggregation operation can proceed without forcing threads to wait for bank access (see Figure 8). This complex optimization solves all conflicts when accessing memory banks, reducing the execution time to 2.58 ms from 4.48 ms. Without using shared memory, a straightforward implementation consumes 15.40 ms instead (see Table 2).

**Phase 3: LBP operator** The computation of the LBP feature entails a convolution with a 3x3 matrix, followed by a binary to decimal conversion (see Figure 4). Convolution requires careful thread scheduling to use shared memory; the responsibility falls on the
Table 2. Major CUDA optimizations in the image analysis application. Execution times correspond to a single 1Kx1K tile on a single GPU. Phase 1’s communication times are for transferring data from the CPU to the GPU, while phase 4’s communication times are for transferring data from the GPU to the CPU.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description / Optimizations</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Baseline version: Using float3 per color channel</td>
<td>8.49</td>
</tr>
<tr>
<td></td>
<td>Coalescing (Alpha channel inserted) on float3</td>
<td>10.79</td>
</tr>
<tr>
<td></td>
<td>Replacing float3 by uchar (256 threads/block)</td>
<td>2.98</td>
</tr>
<tr>
<td></td>
<td><strong>Using shared memory (256 threads/block)</strong></td>
<td><strong>2.98</strong></td>
</tr>
<tr>
<td></td>
<td>Using between 169 and 192 threads/block</td>
<td>2.98</td>
</tr>
<tr>
<td>2.</td>
<td>Baseline version: Using global memory</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Using shared memory for co-occurrence matrices</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td><strong>Solving conflicts on shared memory banks</strong></td>
<td><strong>None</strong></td>
</tr>
<tr>
<td>3.</td>
<td>Baseline version: Special threads on grid borders</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td><strong>Blocks of 16x16 threads, 14x14 computing</strong></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Blocks of 8x8 threads, 6x6 of them computing</td>
<td>None</td>
</tr>
<tr>
<td>4.</td>
<td>Baseline version: Using global memory</td>
<td>4.02</td>
</tr>
<tr>
<td></td>
<td>Using shared memory for local histograms</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td><strong>Solving inter-warp conflicts in memory banks</strong></td>
<td><strong>0.31</strong></td>
</tr>
<tr>
<td>Total</td>
<td>Baseline version</td>
<td>12.51</td>
</tr>
<tr>
<td></td>
<td>Involving shared memory</td>
<td>3.29</td>
</tr>
<tr>
<td></td>
<td><strong>Optimal version</strong></td>
<td><strong>3.29</strong></td>
</tr>
</tbody>
</table>

Thread deployment within blocks. Each thread requires 10 registers and each block of threads uses 296 bytes of shared memory. Due to the memory usage characteristics, we were able to allocate 256 threads in a 16 x 16 grid to reach 100% occupancy on the G80. Each thread reads a pixel from global memory and stores it in a shared memory data structure. Those threads located on the border of the grid are unable to compute, since they do not have access to their neighbor data (see Figure 9); they exit the kernel at this stage.

The LBP for the border regions will be computed by the next block of threads, since we overlap the thread layout by two rows and two columns of pixels each time. By using this strategy, we incur 23% idle cycles and the associated redundant accesses which increases memory contention, but the computation is now more homogeneous and the threads are more lightweight. As compared to the heterogeneous version with no idle cycles and no memory redundancy, the homogeneous version is 25% faster, leading to the lowest execution time of 1.82 ms.

Since the LBP kernel is a very regular computation, we can select different sizes of thread deployments, provided a square grid is used. Here, we investigated the effect caused by different thread/block ratios. We gathered values for grids of threads of sizes 20x20 (2.02 ms), 18x18 (1.90 ms), 16x16 (1.82 ms - optimal), 14x14 (1.89 ms), 12x12...
(2.00 ms), 10x10 (2.16 ms), and 8x8 (2.31 ms - reported in Table 2 as the last version for phase 3). A grid of 16x16 threads is a popular choice among expert programmers to maximize performance in CUDA; that said, our intention was more to quantify the penalty we may incur by making an inappropriate choice for thread allocation.

**Phase 4: Histogram**  The implementation of this phase is based on a histogram kernel included with the CUDA library [17], where the global reduction is decomposed into two phases. The first phase, performed by the GPU, accumulates values within all the threads of a block (which ends up packing results in a single and distinctive value for each block). The second phase, which accumulates the block’s partial histogram values, has to be delegated to the CPU since there is no mechanism for communicating between threads in different blocks.

We deemed this an appropriate solution, since the histogram is only a marginal phase in our application; it is only computed once per tile, and it has a light workload compared to the rest of the phases. The execution time for this last kernel is 0.9 ms.

Table 2 summarizes all of the major optimizations performed within CUDA on each phase along with the execution times obtained. Since the current CUDA programming model does not allow the developer to assign different kernels to different multi-processors, the four phases are sequentially executed and the whole GPU is used during each phase.

Overall, by using CUDA to implement the image analysis, we were able to reduce the execution time by a factor of 3-5 compared to the times obtained by Cg (see Figure 11 for times on an entire image), by an additional factor of 3 when enabling shared memory, and by an extra 20% when solving memory conflicts as much as possible for an optimal execution.

Finally, it is remarkable how the potential improvements by solving shared memory bank conflicts are quite modest when compared to just making use of shared memory. This is particularly interesting if we consider that the former task is more challenging and time consuming for the programmer in most of the applications.

2.2. **Stream Programming with DataCutter**

This section details why the streaming programming model is appropriate for large-scale scientific applications. It details why DataCutter meets the challenges of these applications and allows developers to write scalable parallel applications making full use of disruptive architectures. Also, we discuss the specific implementation details related to our case-study image analysis application.

The streaming programming model is an excellent choice for large-scale, data-intensive scientific applications which must often process gigabytes of data within seconds [1,12,27]. The streaming model naturally makes use of techniques associated with high performance programming such as data blocking, communication overlapping and bottleneck avoidance. It is only with these high performance techniques that large-scale analysis applications can fit into time-sensitive workflows. The streaming programming model maps the top-level application task graph directly into software components, making even large application development easy to understand and plan. Applications designed using this model are decomposed into sequential tasks (filters) with explicit data dependencies. Data flows from filter to filter through logical streams in a buffered, non-blocking manner. Separating application tasks into reusable components during software
development is a hallmark of well-designed applications anyway; streaming programming therefore is not a large burden for developers, since the stream processing functions often correspond with normal application subdivisions.

DataCutter [2] is a component-based distributed middleware framework providing a coarse-grained dataflow system and allowing combined use of task- and data-parallelism. Each filter in a DataCutter application executes within a separate POSIX thread inside the main DataCutter process. All of the communications between filters residing on the same node can communicate with pointer sharing or the copying of memory. TCP sockets or MPI communications (in order to leverage low latency, high bandwidth interconnects such as Infiniband) are used for communication between filters on different hosts. The buffered nature of stream communication allows for the easy overlap of data communication and computation, and therefore the hiding of disk and inter-node communication latencies.

Since DataCutter is a distributed, multi-threaded middleware framework, it is excellent at leveraging the types of clustered multi-core supercomputers targeted in this paper. The use of technologies such as POSIX threads for intra-node parallelism and MPI for inter-node parallelism ensures the best use of single node and communication resources. Further, the filter-stream programming model eases the use of disruptive technologies such as GPUs. GPU-specific filters can be substituted for more standard C++ (or even Matlab) filters with ease at any point in the development process, provided a common data interface is established.

In general, a small number of techniques used during the development of filter-stream applications is sufficient to guarantee a measure of success. By thinking about how the computation to be performed will be affected by various hardware limitations - such as the memory bandwidth, communication bandwidth, and ALU bandwidth - the developer can decide on the appropriate granularity for the filter tasks. If the tasks are too small, then the overhead related to the movement of data can become burdensome. If the tasks are too large, then insufficient parallelism is exposed. Additionally, ensuring the data structure interface stays consistent can ease the introduction of new filter types and the mixing of these filters during deployment on heterogeneous computer architectures. We will discuss all of these techniques as they relate to the implementation of our case-study application.

Our image analysis application is easily divided into three stages, each of which is implemented as a single filter type. These are: a TIFF-Reader filter that reads binary TIFF tiles, a TIFF-Decompressor filter that decompresses TIFF tiles and produces RGB images, and a Tile-Analysis filter where the real image analysis computation is carried out. Since the image analysis algorithm is designed to operate on image tiles, the most natural application subdivision (which we chose to use) is to keep the tile as an indivisible data item. This allows direct comparison of GPU analysis filters with more standard filters. Since this granularity level gave the application good overall performance, we did not experiment more with other application task divisions.

Figure 10 shows the general layout of the system. One or more reader nodes (with one TIFF-Reader filter each) read the binary TIFF image tiles from the disk and write them to the stream leading to the TIFF-Decompressor. One or more TIFF-Decompressor filters are able to coexist on the same node. This decompressed image data is then written to the input stream to the Tile-Analysis filter. Placing both the TIFF-Decompressor
and the Tile-Analysis filters on the same node has the benefit of saving memory and communication bandwidth.

A filter layout which includes one each of these three filter types constitutes a complete image analysis task graph, while multiple copies of each filter type allows for simple parallel and pipelined execution. For all filter-stream applications, the decisions about how many filter copies to instantiate and how to map them to the computer resources are heavily influenced by examining the hardware bottlenecks for this specific application. These bottlenecks are affected by which filters are chosen for the analysis portion of the application. For instance, the Cg GPU filter can execute the analysis algorithm at a rate such that one TIFF-Decompressor filter is sufficient to allow the GPU to waste no time waiting for data. Therefore, a DataCutter task layout with one TIFF-Decompressor for each Cg GPU filter is sufficient. However, the CUDA GPU filter needs three TIFF-Decompressor filters such that it will not stall. Lastly, by keeping the input data format the same for each Tile-Analysis filter type, we are able to transparently (to the filters upstream, and the end result of the analysis) swap filter types or combine them in a specific application layout as we see fit.

The next section details the results of our experiments with our case-study application on a specific cluster architecture. The various application decisions will be highlighted again in terms of the techniques used to gain the highest level of performance.

3. Experimental Results

Our experiments were performed on the Ohio Supercomputer Center’s BALE cluster. BALE is comprised of 55 nodes with AMD Athlon 64 dual-core CPUs and 18 nodes with dual AMD dual-core Opteron CPUs and dual NVIDIA Quadro FX 5600 graphics cards. The interconnection network for the BALE cluster is Infiniband. Our experimental setup uses 16 GPU-equipped nodes as computation nodes and an additional six nodes as reader nodes, in order to provide enough disk bandwidth to avoid the disk I/O bottleneck. These
reader nodes had their system file caches preloaded with several discarded executions to ensure extremely high I/O from the upstream, TIFF-Reader filters. We feel this is a suitable experimental setup to use; any production cluster designed to execute this type of imaging application with multiple GPUs and a fast interconnect can reasonably be said to have parallel disks providing high I/O bandwidth.

In our experiments, we have used three different digitized pathology images. Table 3 summarizes their features.

<table>
<thead>
<tr>
<th>Name</th>
<th>Resolution in pixels</th>
<th>Number of 1Kx1K tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALL</td>
<td>32,980 x 66,426</td>
<td>33 x 65 = 2,145</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>76,543 x 63,024</td>
<td>75 x 62 = 4,659</td>
</tr>
<tr>
<td>LARGE</td>
<td>109,110 x 80,828</td>
<td>107 x 79 = 8,453</td>
</tr>
</tbody>
</table>

The first experiment sets the baseline single node CPU and GPU performance for the various implementations of the image analysis algorithm. Figure 11 shows the algorithm execution time and overhead time of each implementation when analyzing the SMALL image. The first four stacked bars represent CPU-only implementations, while the last six stacked bars bring one or two GPUs into the fold. Those bars with labels beginning with ‘DC’ are results from those implementations using DataCutter and those without the ‘DC’ label are the basic, serial implementations.

Figure 11. Execution time comparison of all implementations of the image analysis codes running on a single node using SMALL image.

The execution time in Figure 11 (shown by the lower portion of each bar) is due solely to the actual image analysis, while the overhead (shown by the upper portion of each bar) is caused by disk I/O, TIFF decompression, remote process invocation, and network latencies, where applicable. In this experiment, the most important thing to note is the two to three orders of magnitude of performance speedup when moving from the
CPU-based solutions to the GPU-based solutions. Additionally, the DataCutter versions of the GPU-based image analysis algorithms are able to shorten the execution time for the entire image versus the non-DataCutter versions, since the decoupled, multi-threaded nature of DataCutter allows the image analysis to overlap with the TIFF tile decompression and the disk I/O. Further, the overhead time for DataCutter implementations is less than that for the serial CPU or GPU implementations, since DataCutter can leverage the multiple multi-core CPUs on the BALE nodes and run more than one TIFF-Decompression filter per node. Unfortunately, the CUDA implementation of the image analysis algorithm is fast enough to cause the TIFF tile decompression stage to become a bottleneck when two GPUs are used. This stalling prevents both GPUs from being fully utilized. Lastly, since four C++ threads shows a clear advantage over running a single thread, all future C++ results will be comprised of the DataCutter version with four tile analysis threads per node.

Figure 12. Execution time comparison of GPU and DataCutter implementations running on a single node using all three input images.

Figure 12 shows the single-node performance comparison of the GPU-based implementations for the three images. The main point to take from this chart is that there is a linear relationship between the execution time and the overall size of the image under analysis in all of the implementations. Unfortunately, even when analyzing large images, making full use of two GPUs is hindered by the inability for the TIFF decompression stage to keep up; this being the case, we will not show dual GPU results for the remainder of the experiments.

Figures 13, 14 and 15 show the scalability of our solution with respect to the number of nodes. As in Figure 12, the lower portion of each bar represents the image analysis time, while the upper portion of each bar shows the aggregated overhead. This type of image analysis computation scales extremely well, resulting in image analysis execution times which decrease nearly linearly with the number of nodes. Further, the total analysis times for the DataCutter/CUDA implementation are under four seconds for the SMALL image, under seven seconds for the MEDIUM image, and just over eleven seconds for the LARGE image, when running on sixteen nodes. Compared with the single node CPU C++ computation time of nearly an hour for the SMALL image, this represents a tangible
Figure 13. Parallel execution times and speedups of C++, Cg, and CUDA based DataCutter implementations using SMALL image while varying the number of nodes from 1 to 16.

Figure 14. Parallel execution times and speedups of Cg, and CUDA based DataCutter implementations using MEDIUM image while varying the number of nodes from 1 to 16.

benefit of increased productivity. Since the main focus of this paper is the GPU results, and since the C++ result is shown to scale well in the worst-case (because it incurs the lowest proportional overhead), we have chosen to remove it from the figures showing results for the MEDIUM and LARGE images.

As seen in the speedup figures, there is good speedup since the tiles are able to be decompressed and processed entirely independently of each other. However, due to the small execution times in the GPU-based implementations, the various overheads (comprised of remote process startup, network, and TIFF decompression latencies) begin to become comparable in overall time to the total time spent per node processing the image tiles. For instance, on sixteen nodes, the CUDA implementation requires at most 1.80 seconds of computation to compute 2,145 tiles (the number in the SMALL image. However, despite concurrently running three TIFF-Decompressor filters, the decompression time alone for each node’s allotment of 135 tiles could range from 0.3 seconds to 1.3 seconds.

Nearly linear speedup could be achieved in a production environment, however, since it is reasonable to assume that remote process invocation would only occur once for many images which are to be analyzed. Under these server-like circumstances, only the I/O system and network latencies would comprise the system overheads.
In this paper, we have presented design trade-offs and a performance evaluation of a sample biomedical image analysis application running on a cooperative cluster of CPUs and GPUs.

By implementing algorithms on GPUs using CUDA and using DataCutter to parallelize the computation within and across nodes, we establish techniques to fully exploit the parallelism inherent in the target heterogeneous, distributed architecture and in the application: multi-node (using DataCutter for data partitioning across nodes), SMP and thread-level (using DataCutter to fully utilize the available on-node and on-chip hardware resources), SIMD (using CUDA to fully populate the 128 stream processors of the GPU with work), and finally, ILP (Instruction Level Parallelism, by setting up blocks of computational threads within the GPU execution).

Our experimental results show great success for our techniques, first by decreasing the execution time on a single node by using different optimizations, and then extending those gains to multiple-node parallelism for a scalable distributed execution. On a single node, we obtained up to 46.5 times speedup, in end-to-end application, using CUDA over optimized sequential C++ code. By using two GPUs per node, the single-node time to process the image is under one minute, if you ignore the overheads associated with disk I/O and tile decompression, proving that the CUDA method is extremely powerful. Additionally, the use of DataCutter to overlap the computation with disk I/O and tile decompression helps the GPU stay as busy as possible, both on a single node as well as on a multi-node setting. When analyzing the largest test image and including overheads, on the 16 node cluster configuration, the fastest DataCutter-CUDA implementation is 35.1 times faster than the serial CUDA implementation. Hence by using 16-node GPU cluster, in total, we achieved nearly three orders of magnitude speedup (about 926 times with our large test images) in end-to-end application, compared to optimized C++ code. Overall these optimizations, brings the computation time of a single large tissue image under 12 seconds and makes this implementation usable in a clinical setting. Should additional computational resources be available, it would be straightforward to leverage them by independently analyzing 5-6 slides of each biopsy sample.

GPUs are highly scalable and are becoming more powerful and applicable for general purpose computing; we envision biomedical image processing as one of the most exciting fields able to benefit from the use of GPUs. Additionally, new tools like CUDA may assist non-computer scientists, providing an easier GPU programming interface for...
adapting biomedical applications. By combining this computational power with DataCutter to parallelize the computation across clusters of GPUs, a variety of scientific applications can give near real-time responses to questions of all kinds.

Acknowledgements

We thank Dennis Sessanna and Don Stredney at the Ohio Supercomputer Center for support running our code on the BALE visualization cluster.

References


System Level Accelerator with Blue Gene: A Heterogeneous Computing Model for Grand Challenge Problems

Tim DAVID a,1

a Centre for Bioengineering, University of Canterbury, Christchurch, Canterbury 8020, New Zealand

Abstract. An algorithm is presented which can be used to "link" two differing computer architectures together in what has become known as System Level Acceleration (SLA). This allows for a substantial increase in the complexity of the problem to be solved and also in the range of scales that often inhibits the scientist/engineer. Two examples of problems able to be solved via SLA, that of coupled cells in an vascular domain and modeling blood auto-regulation in the cerebro-vasculature.

Keywords. System level acceleration, cerebro-vascular blood flow

Introduction

Supercomputing architectures have evolved over the past ten years on essentially two tracks. Firstly those of the SMP cluster type architecture (ie the IBM p-series and the Cray XMT-series) where SMP nodes are linked together in a cluster-type fashion using high-speed interconnect such as Myranet or Infiniband and secondly the fully distributed node architecture exemplified by the IBM Blue Gene. Both have advantages and disadvantages. Yet up until now both could not be brought to bare on a single problem.

In line with these architectural advances science and engineering have each been posing more and more complex problems which are defined on complex geometric physical spaces. These physical spaces are themselves defined over vast ranges of scale lengths as will be shown in the following sections. In order to solve problems whose scale lengths vary substantially there are two possible solutions. Either discretise down to the smallest scale with the possibility of producing such large data sets and numbers of equations that the memory requirements become too large for the machine or divide the problem into a subset of appropriate length scales and map these discretised sub-domains onto appropriate machine architectures. The definition of "appropriate" here is determined on a case-by-case basis at present. However an example will be described below.

1Corresponding Author: Director, Centre for Bioengineering, University of Canterbury, Christchurch, Canterbury 8020, New Zealand; E-mail: tim.david@canterbury.ac.nz
1. The Multiple Scales Problem

There are a significant number of problems that exhibit a large range of physical scales, for example small vortex generators positioned on large scale aerofoils; but none so prominent in the 21st Century as that exemplified within the biological sciences and engineering. Biological Engineering problems have a multitude of physical scales. In the major arterial networks the blood flow dynamic scales are of the order of 1mm (cerebral vessels) up to 25mm (ascending aorta). Downstream of any major vessel exists a substantial network of arteries, arterioles and capillaries whose characteristic length scales reach the order of 10-20 microns. Within the walls of these cylindrical vessels lie ion channels consisting of proteins (100 nanometers and smaller) folded in such a way as to allow only certain molecules through the membrane. One can now of course ask the question as to why all these scales should be integrated into a single model. Below we outline just two examples.

1.1. Modeling the Cerebral Vasculature and Atherosclerosis

For some years the biological, physiological and engineering communities have been trying to understand vascular disease, in particular atherosclerosis, a chronic and inflammatory response in the walls of arteries. This is due to the accumulation of macrophage white blood cells and promoted by low density lipoproteins (plasma proteins that carry cholesterol and triglycerides) without adequate removal of fats and cholesterol from the macrophages by functional high density lipoproteins (HDL). It is commonly referred to as a "hardening" or "furring" of the arteries and is caused by the formation of multiple plaques within the arteries. But this does not necessarily answer the question. Some twenty years ago research results started to indicate that atherosclerotic plaques occurred at specific sites in arteries and that blood flow dynamics seemed to play an important part. So the large physical scales were interacting in some way with the smaller scales of the cellular chemistry. It became clear that at the interface between artery wall and blood flow was not a passive boundary but an active organ that communicated large scale dynamics of blood flow (primarily the tangential fluid shear stress on the cell) to the cellular chemistry of the endothelium and beneath it to the underlying smooth muscle cells and cellular matrix. It was clear that if we wanted to fully understand the growth of atherosclerotic plaques both ends of the length scale needed to be modeled.

Cerebral tissue requires a constancy of both oxygen and nutrients (notably glucose). During periods of pressure variation, which occur throughout the normal day as well as in cases of pathological hypo- and hyper-tension, the body’s cerebral autoregulation mechanism cause the arterioles to vasoconstrict/dilate in response to changes in cerebral perfusion pressure over a certain range, thus maintaining a relatively constant cerebral blood flow. These effects are of particular importance when investigating how blood is redistributed not only via the circle of Willis, but throughout the cerebral tissue. The circle of Willis (named after Thomas Willis, an English physician) is a set of linked (in a circular topology) arteries located at the base of the brain. This circular structure enables both hemispheres of the brain to be perfused with blood should one or a number of incoming arteries become partially or fully blocked. It should be noted that there have been a number of cerebral autoregulation models proposed [1,2,3], including models incorporated with a circle of Willis [4,5,6,7,8,9,10,11]. Because the resistance of the arterial tree...
varies as the 4th inverse power of the radius of an artery the majority of resistance to flow occurs in the smaller vessels. To investigate the way in which the brain responds to variations in pressure and yet maintains a virtually constant supply of blood to the tissue numerical models need to be able to have a representation of not only the tree but also a dynamic model of how the small arteries constrict and dilate. Simulating this phenomenon as a “lumped” connection of arteries is insufficient since different parts of the arterial tree respond differently. Thus we have a range of scales from the major arteries (of the order of mm) down to the arteriolar bed (of the order of microns). However, the combination of a 3D model taken from MR data coupled with an autoregulation model with a fully populated arterial tree able to regulate dynamically remains a relatively unexplored field. Having now given a brief (if not possibly unduly oversimplified) explanation of some very complex biomechanics we look at the difficulties involved in developing a system which can simultaneously solve over all length scales.

1.2. The numbers problem

To appreciate the range of scale lengths and the consequences of trying to solve the entire problem on a single architecture we continue with two vascular/arterial examples.

1.2.1. Cerebro-vasculature

Experiment seems to indicate that about 35 mm$^3$ of cerebral tissue provides on average some 360,000 individual vascular segments [12]. In order to cover a volume whose size is of the order of the resolution of say an MR (magnetics resonance) image, for validation proposes then each volume (or voxel in this case) would need to be modeled by approximately 400,000 segments. For the whole brain, taking into account the heterogeneity of the tissue where the cortex is far more vascularized than the white matter the total number of vascular segments would of the order of $4 \times 10^{10}$!

Each one of our vascular tree models, using the work of [13], (a sketch of one section of the tree is shown in Figure 1) are such that they contain approximately 1,000,000 segments with 500,000 terminal leaves (representing the smallest arteriole). Each leaf has a number of capillaries which it perfuses up to 6,000 - 10,000 capillaries. These capillary numbers have been determined based on the volume of perfused tissue for the middle cerebral artery and an assumed capillary radius of $10 \mu m$.

1.2.2. Atherosclerosis, the Endothelial cells and the major/minor arteries

Figure 2 shows a section of a porcine arterial system [14]. For each artery of 1mm diameter and 20 mm long there are approximately 12,000 Endothelial cells. For the arterial geometry shown in Figure 2 there are 1.8 million cells and underlying them are 2 million smooth muscle cells (SMCs).

As noted above the endothelial cells and smooth muscle cells are coupled such that chemical species diffuse across the cell membranes. They effective talk to one another and share information about the blood flow and chemical concentration of not only the blood but also the concentrations in the cells themselves. This means that any model needs to represent this “coupling” and to pass information back to the higher scales of blood flow.
1.3. System Level Accelerators: A Hybrid Computing Solution

We need to find a computing architecture which supports the modeling of a range of scales exemplified by the examples given above. Specifically the architecture should support
- binary tree topology with simple communication "up" and "down" the tree
- nearest n’b’d communication (cell coupling)
- Domain decomposition for mixed parabolic/elliptic equations (Navier-Stokes fluid flow equations)

all of which needs to be integrated into a single system to solve a single problem.

As Drahzal et al clearly state [16] it seems that the "classical CPU scaling" is nearing an end for two major reasons. Firstly that of the ratio of active to passive power (a ratio that shows the relationship between the power used to compute and the power required of the processor to enable it to compute). Secondly that of energy consumption. As is pointed out scaling solutions in a cluster configuration "makes the energy consumption
issue even worse” since it requires a considerable additional bureaucracy of fans disk drives etc to support the cluster. The numerical solving kernels of a large majority of 3D computational fluid dynamics software have been written in FORTRAN although a notable exception to this is OpenFOAM (see http://www.opencfd.co.uk/openfoam/). Independent Software Vendors cannot afford to rewrite existing HPC code when the “need for speed” is at the moment at it’s greatest however, System Level Acceleration (SLA) allows multiple, disparate computer systems to complete a computational problem. At present there are two types of system level accelerator, the loose coupled (LCA) and the tightly coupled acceleration (TCA) systems. The LCA is a well used accelerator where there exist separate workstreams, such that the output from one is stored for future use by another workstream. Figure 3 shows the host and accelerator coupled via the scheduler and parallel file system. For the large problems that require solving there is a significant amount of disk file storage required and a large amount of time spent reading and writing to disk. A common example would be the solution methods relating to fluid/structure interaction where one workstream is a fluids code whilst the other a solid stress code.

![Figure 3. The basic scheme for a loosely coupled accelerator (LCA)](image)

For our two specific problems we use a TCA as outlined below.
2. Grand Challenge Problems: Fluid-cell coupling and Autoregulation in the Cerebro-vasculature

2.1. Fluid-cell coupling

For this particular problem we need to determine the action on the endothelial cells and underlying SMCs by the flowing blood. The blood exerts a tangential force on the endothelial cells which subsequently evokes complex cellular reactions (e.g. calcium flux into and out of the cell). These reactions elicit a response in the smooth muscle cell with one of a number of candidates for atherogenesis that of a reduction or variation in Nitric Oxide. To complicate even further the smooth muscle will contract and the endothelial cell will exude agonists into the flowing blood which forms a feedback loop via the blood fluid dynamics. Full details of the theory and algorithm can be found in [14]. Figure 4 shows the connections between ECs and SMCs. In order to complete the algorithm that takes into account the large numbers of cells, the cell-cell coupling and the fluid dynamics we divide the problem up into the following sections.

Figure 4. Basic sketch of the smooth muscle/endothelial cell connections

- **Fluid flow and species convection**: we solve the non-dimensional incompressible Navier-Stokes equations given in cartesian co-ordinates by

\[ \frac{du}{dt} + u \cdot \nabla u = -\nabla p + \frac{1}{Re} \nabla^2 u \]  

(1)

and the conservation of mass as

\[ \nabla \cdot u = 0 \]  

(2)

and finally the conservation of species

\[ \frac{d\phi_i}{dt} + u \cdot \nabla \phi_i = \nabla^2 \phi_i + R_i \]  

(3)
The discretisation algorithms for equations (1), (2) and (3) are not detailed here as there are a number of ways this can be done. The solution algorithm is implemented on a SMP cluster type architecture (in this case an IBM p-575). We do this for several reasons. Firstly because the fluid domain (which includes the domain for the chemical species flowing in the blood) can be decomposed into subdomains and each subdomain is mapped onto a separate processor. Although this requires essentially a matrix inversion of the Jacobian of the system equations of solution variables the boundary communication between subdomains is substantially smaller than dividing up into singular elements. Secondly the fluid dynamic equations are mixed parabolic/elliptic and thus scale reasonable well on this type of architecture.

- **Cellular chemistry** : In order to understand the mechanism/processes by which the vascular wall responds to hemodynamic changes, a set of nine ordinary differential equations is used to model the coupled endothelial and smooth muscle cells of the wall. The calcium dynamics in a single endothelial cell is modeled by a set of 4 ODEs with variables, cytosolic calcium concentration, calcium concentration in ER, membrane potential and cytosolic inositol 1,4,5-trisphosphate (IP3) concentration. The IP3 concentration is proportional to the concentration of agonists in the blood (derived from the fluid dynamics equations solved on the p-575). Agonist binding to ligand specific receptors on the cell plasma membrane can initiate production of intracellular IP3 via the phospholipase C (PLC) pathway, thus work as a stimulus to intracellular calcium increase. In addition the shear stress exerted on the endothelial cell by the flowing blood provides an added influx of calcium via shear stress mediated ion channels. This wall shear stress is again derived from the fluid dynamics equations solved on the p-575 and hence a boundary value required to be communicated between the Blue Gene and the p-575.

Cytosolic calcium concentration is an important entity which modulates the tone of an excitable cell such as a smooth muscle cell and is proportionate the force of contraction. In the resting state, the cytosolic calcium concentration is much less (0.2μM) compared to extracellular fluid (2μM). Various active and passive processes work towards maintaining it by either extruding calcium out of the cell or refilling the intracellular calcium stores through Sarcoplasmic Reticulum Ca-ATPase (SERCA) pump. Upon binding of IP3 to specific receptors on sarcoplasmic reticulum (SR), the calcium from these stores is released into the cytosol. Increase in cytosolic calcium can induce further calcium release from the SR stores via ryanodine receptors on the SR surface, a process called Calcium induced calcium release (CICR). Increase in cytosolic calcium concentration depolarizes the membrane and allows influx of extracellular calcium through Voltage Gated Calcium Channels (VOCC) of the plasma membrane, further increasing the calcium inside the cell. In addition, Na/Ca exchanger also brings in calcium from outside. Increased calcium ions binds to calcium specific binding sites on the cytoplasmic side of the Calcium activated Potassium channels (KCa) in the plasma membrane, resulting in activation of outward positive current to restore the resting membrane potential. VOCC also close due to repolarization of the membrane potential thus decreasing the rate of influx of extracellular calcium. Simultaneous to all these processes, Ca^{2+} is continuously extruded by Ca-ATPase in the plasma membrane and pumped back in the SR by the Ca^{2+} dependent low affinity SERCA pump. This has been model as a system of five ODEs for a smooth muscle cell. The five time varying variables are, cytosolic calcium concentration, calcium concentration in SR, membrane potential, probability of open K^+ channels and cytosolic IP3 concentration. The agonist concentration (stimula-
tion) is proportional to the rate of PLC production and thus to the rate of intracellular IP3 generation. We couple the endothelial and smooth muscle cell by including a conductance term that allows both membrane potential and chemical concentration to “diffuse” via gap junctions to neighbouring cells. A system of ODEs for the coupled single smooth muscle cell and the endothelial cell was solved numerically using 4th order Runge Kutta integrator, written in in C++. We define a unit to be a set of coupled cells (made from a number of endothelial cells and a number of smooth muscle cells). Each unit can therefore be mapped onto a BG node (essentially a single processor where the other processor is retained for communication purposes). For this part of the solution algorithm we require only nearest neighbourhood communication since the cells communicate only with their neighbours. The IBM Blue Gene architecture is particular suited to this type of domain splitting. Periodic boundary conditions (due to the circular nature of the vessel geometry) is easily accomplished on the Blue Gene due to its inherent topological structure. Each unit (essentially a block of coupled cells) is logistically and geographically linked to its nearest neighbour unit. We utilise in the algorithm a communication between each Blue Gene node that provides (at each time step of the integration) the required membrane concentration and chemical species concentration along the boundary of each unit.

- **System Coupling:** The fluid equations require boundary conditions for both species concentration and blood velocity at the cell boundary. This is exactly where the boundary between the differing computer architectures have been defined and an interesting question could be posed here in that should the computational boundary between differing machines be defined by the (internal) boundary conditions of the problem, i.e. are computational boundaries a generic hardware boundary? The actual coupling can be achieved in the following manner. As is readily seen we require a Tightly Coupled Accelerator. The p-575 requires to have information on cell concentrations, artery wall motion, whilst the Blue Gene requires the fluid shear stress and agonist flux mapped to each cell unit. We will utilise an RPC (Remote Procedure Call) for this particular cell-fluid coupling. At each time-step boundary information of fluid shear stress and agonist concentration is transferred from p-series to BG whilst, since cells exude agonists themselves then, boundary data on the efflux of agonists from each cell unit is communicated from the Blue Gene to the p-575. Care must be taken to map correct cell values to the correct geographic place in the fluid domain. Hence the RPC contains not only variable values but important information concerning the functional mapping from one architectural domain to another. after each exchange of information the p-575 uses the new boundary data to solve its fluid flow equations whilst the Blue Gene uses its new data to evaluate the complex chemistry variations in each of the cell units. This continues until a set number of time steps has passed. Communication between the p-575 and the Blue Gene is only done at a priori specified times during the solution. This is due to the realization that the adaptive time steps used by both integrators may be incur different time step increments. The basic algorithm is given below

1. set initial conditions for fluid domain and arterial wall cells.
2. at time step n assume a solution in the fluid domain for pressure, velocity and ith species concentration of $p^n, q^n$, and $\phi_i^n$ and in the arterial wall cells of species $\psi_j^n$
3. using $q^n$ find the wall shear stress vector $\tau_w$ and the agonist concentration $\phi_i^p$ at the arterial wall.
4. pass $\tau_w$ and $\phi_i^n$ values to the Blue Gene nodes.
5. pass agonist flux $\phi_j(w)$ from the arterial wall to the p575.
6. For $\Omega_{3D}$ solve equations (1), (2) and (3) using new agonist flux $\phi_j(w)$ as species boundary condition for the fluid domain.
7. using $\tau_w$ and $\phi_i^n$ as input values solve the set of o.d.e. s modeling cellular chemistry.
8. go to 2 and repeat until convergence.
9. advance the time step until a specified time or an asymptotic solution is reached.

2.2. Autoregulation in the Cerebro-vasculature

For the second example we choose a more "simple" example, that of blood flow in the brain. The ultimate goal here is to model blood perfusion (flow) in the entire cerebro-vasculature, an exacting task given the numbers of vessels that we calculated earlier! Both experiment and computational modeling has shown that we must look at both the major arteries in the brain as well as the vast arterial structure brilliantly exemplified by Lauwers et al [12]. Figure 5 shows an example of our "in-house" software which extracts (segments) arterial information from MR data. This data provides the fluid domain geometry on which we solve the 3D Navier-Stokes equations in the same way as explained in the above section.

![Figure 5. segmented 3D data from Magnetic Resonance images of the major cerebro-vasculature](image)

The tree branching algorithm used is that developed for the abdominal fractal vascular network of Olufsen et al [13] and is based on two variables: a power exponent $k$ (describing the relationship between parent vessel radius $r_p$ and daughter vessels radii $r_{d1}$ and $r_{d2}$, and an asymmetry ratio $\gamma$ (describing the relative ratio between two daughter vessels). For this particular study a binary tree emulates the vascular system stemming
from a major artery, say the middle cerebral artery (MCA) for example. The values of $k$ and $\gamma$ change depending on the location of the vessel in the tree-i.e. values are different for arteries, arterioles and capillaries. The length and radius of a vessel are related by a length to radius ratio (denoted $l/r$, chosen as 20 for this model). Here we take values of $k$ and $\gamma$ as those given in [13]. The arterial tree code creates branching levels. At each branching level, two daughter vessels are created for each artery in the previous branching level. This process is repeated until all terminals reach the size of a pre-capillary sphincter ($10\mu m$). Each of these terminals is assumed to supply a bed of capillaries, all with $2.5\mu m$ radius. Figure 6 shows the number distribution of terminal arterioles as a function of pressure and indicates a normal distribution correlating well with the data of Lauwers et al [12].

The vascular tree whose scale lengths are too small to be seen by the MR are created using the algorithm described above. There are $N$ of these trees where $N$ is the number of major efferent arteries emanating from the circle of Willis. There are approximately 6-10 efferent arteries created from the MR data depending on the depth of resolution. In order to provide the correct information from the vascular tree to the 3D Navier-Stokes solver we require the data exchange of

- outlet flow from the 3D domain
- peripheral resistance of the vascular tree

These two data are sufficient to find the unknown pressure at the boundary interface between the two computer architectures. The basic algorithm is shown in Figure 7 and consists of
1. Evaluate current resistance of vascular tree, \( R \) as described below.
2. At time step \( n \), assume a solution for pressure \( p \) and flow \( q \) in \( \Omega_{3D} \) and \( \Omega_{0D} \) of \( p^n, q^n \) and \( P^n, Q^n \) respectively.
3. For \( \Omega_{3D} \), solve equations (1) and (2) using \( p^n_0 = P^n(a^+) \) as downstream boundary condition for 3D.
4. From the velocity profile \( v^n_0(a^-) \) at the outlet of \( \Omega_{3D} \), evaluate fluid flux outlet \( q^n_0 = \int_v v^n_0(a^-) \cdot \hat{n} d\sigma \).
5. For \( \Omega_{0D} \), let \( q^n_0(a^-) = Q^n_0(a^+) \) and solve \( P^n_1(a^+) = Q^n_0(a^+)R \).
6. Let \( p^n_1(a^-) = P^n_1(a^+) \) and repeat from 2 until convergence.
7. For \( P^n, Q^n \) using myogenic and metabolic models, find new radii of the vessels in the vascular tree and evaluate a new resistance \( R \) of the tree.
8. Advance the time and go to 2 until a set value is reached or asymptotic solution is obtained.

**Figure 7.** Interface boundary condition for the TCA algorithm

To find the total vascular tree resistance \( R \) each individual resistance of each segment in the tree, the algorithm uses a Poiseuille flow assumption such that the resistance of an arterial segment is proportional to the inverse of the fourth power of the radius. The viscosity is calculated using the diameter dependent equations from Pries and Secomb [17]. In order to find the flow and pressure distribution throughout the tree, a single pass up and down the tree is required. Starting at each terminal segment, a total resistance is evaluated for the two daughters and the parent artery assuming that the terminal segments are connected via the venous bed (essentially daughters in parallel with the parent in series). This parallel-series calculation is then done at each bifurcation of the tree until there is a single value of resistance. The total flow into the arterial tree is known and hence the total pressure drop can be evaluated. Moving down the tree, the pressure drop and flow rate at and through each arterial segment is calculated respectively.

The myogenic and metabolic models which evaluate the vessel radius of each segment will be a number of odes dependent on the complexity of the autoregulation model used. In our models, we use 8 odes per segment.

The myogenic response for a system of small arteries of the brain modeling an arterial bed can be represented by the system of dimensionless differential equations using...
the formulation and reasoning of Gonzalez-Ferrandez and Ermentrout [18] termed the GFE model. Essentially the myogenic model consists of evaluating the cellular calcium, $Ca_i$ and membrane potential $v$ using the following formal equations

$$C \frac{dv}{dT} = - \left[ g_L (v - v_L) + g_K n (v - v_K) + m_\infty (v - 1) \right] \tag{4}$$

The non-dimensional time rate of change of cytosolic calcium is given as

$$\frac{dCa_i}{dT} = - \rho \left[ m_\infty (v - 1) - k_C Ca_i \right] + \gamma F (\tau_w) \tag{5}$$

where $g_L$ and $g_K$ are ion channel conductances, $m_\infty$ and $n$ ion channel open probability functions and $\rho$ the non-dimensional equivalent of that given in the GFE model and the function F is taken from the work by Wiesner et al. [19], determined from assuming that the shear-dependent influx to be proportional to the fraction of open Ca$^{2+}$ channels (which has a Boltzmann dependence on the strain energy density in the membrane). Details may be found in [14].

The GFE model utilises a combination of parallel, $x$ (denoting the circumferential distance of the artery wall), and series $u$ and $y$ components to simulate the complete motion of the arterial wall, essentially a Maxwell unit. The $y$ unit is contractile and dependent on the amount of cross bridges (and hence $Ca^{2+}$) whilst that of $u$ and $x$ are purely elastic. The rate of change of circumferential distance (essential the radial width of the artery) is given by the sum of forces acting on the Maxwell unit. These are due to the non-dimensional systemic blood pressure $f_{\Delta p}$, the restoring force due to the two elastic components $f_{\Delta x}$ and $f_{\Delta u}$. In it’s non-dimensional form the time rate of change of the arterial circumference $x$ is given by

$$\frac{dx}{dT} = \frac{1}{\zeta} (f_{\Delta p} - f_x - f_u) \tag{6}$$

with

$$\zeta = \frac{0.8 \sigma^\\circ \ C (1 - R)}{\tau g_C a \pi (1 + R)} \tag{7}$$

where $\sigma^\\circ$ is the hoop stress corresponding to maximum muscle activation [20] and $\tau$ a characteristic time for smooth muscle cell relaxation. The hoop stress force due to pressure $Deltap$ is given by

$$f_{\Delta p} = \frac{1}{2} \Delta p \frac{(1 + R)}{(1 - R)} \left[ x \ - \ \frac{0.8 (1 - R)}{x \ (1 + R)} \right]$$

and for this particular model we use $R = 0.6$ [21]. $f_x$ and $f_u$ are the same as that given in the GFE model as is the equation for the time-rate of change of $y$. The equation for $u$ is a simple conservation of length $x = y + u$.

The metabolic response model uses the simple assumption that the set of arterioles feeding the capillary bed are in close proximity to the venous return. Excess carbon dioxide in the venules is diffused to the arterioles and induces a relaxation/contraction of the arteriolar radius thus allowing increased/decreased blood flow to convect away
carbon dioxide and hence maintain the correct CO$_2$ concentration. This model was also used in the 1D models of [22]. Equation 9 represents a conservation equation where the rate of change of carbon dioxide is balanced by the production (due to metabolism) and that convected away by the blood flow.

$$\frac{dCO_2,\text{tissue}}{dt} = CMRO_2 + CBF \left( CO_2,\text{artery} - CO_2,\text{tissue} \right) \quad (9)$$

where $CO_2,\text{tissue}$ is the tissue concentration of carbon dioxide, $CO_2,\text{artery}$ is the arterial concentration of carbon dioxide (assumed to be 0.49ml/ml) $CMRO_2$ is the cerebral metabolic rate of oxygen consumption (assumed to be a constant 0.035ml/g/min for all parts of the brain), which due to the stoichiometry of the aerobic metabolism in the brain tissue, is equal to the cerebral metabolic rate of carbon dioxide production. $CBF$ is the flow rate in the terminal artery and thus entering the capillary bed. We use a reverting differential equation for the radius of the arteriole given by

$$\frac{dr}{dt} = G \left( CO_2,sp - CO_2 \right) \quad (10)$$

here $CO_2,sp$ is the solution to the homogeneous equation for $CO_2,\text{tissue}$ (9). We choose the following non-dimensionalisation

$$c = \frac{CO_2,\text{tissue}}{CO_2,\text{artery}}; t = \frac{T}{CBF_0}; \chi = \frac{r}{r_0} \quad (11)$$

where $r_0$ is the arterial radius and $CBF_0$ is the value of the blood flow in the terminating arteriole at normal physiological conditions (known quantities). With the above equations (9) and (10) become

$$\frac{dc}{dT} = \left( \varepsilon - 1 \right) + Q \left( 1 - c \right) \quad (12)$$

and

$$\frac{d\chi}{dT} = -\xi \left( \varepsilon - c \right) \quad (13)$$

with

$$Q = \frac{CBF}{CBF_0}; \varepsilon = 1 + \frac{CMRO_2}{CO_2,\text{artery}CBF_0} \quad (14)$$

and

$$\xi = \frac{GCO_2,\text{artery}}{r_0CBF_0} \quad (15)$$

Further details may be found in [23]. In essence then the SLA algorithm is defined as

- initialisation procedure

* The 3D fluid domain is firstly meshed into an unstructured mesh (normally hex-block elements), divided into subdomains with each 3D subdomain mapped onto an individual processor on the p-series.
The vascular tree associated with each of the major outflowing arteries are created using the algorithm set out above and mapped to the Blue Gene. Care must be taken here to ensure as far as possible equal workload distribution across the nodes.

Pipes/sockets opened (RPC)

Initial conditions (i.e. at \( t = 0 \)) generated for both 3D fluid domain and the vascular trees.

- for each time step
  - equations (1), (2) and (3) are solved in the flowing blood
  - the mean blood flow and pressure distribution throughout the vascular tree is determined by a pass up and down the tree as described previously.
  - the 3D fluid code calls for its next boundary values from Blue Gene, it needs just the pressure at the root of the vascular tree. data exchanged via RPC
  - the 3D fluid solving code receives its data. data exchanged via RPC
  - the vascular tree solving code calls for its next boundary values from p-575. data exchanged via RPC
  - each code now solves its own equations via an inner iteration due to the non-linearity of the defining equations.

- next time step until end of range

3. Conclusion

A significant number of vascular problems are now being solved by utilising a lumped parameter approach where the physiological complexities are "lumped" together via sets of ordinary differential equations which model, for example the whole systemic system including the heart. However this type of modeling process does not provide the detail now required to understand complex fluid/cell processes. Indeed although useful at a number of levels the lumped parameter model has now been superceded by the necessity of furthering understanding at the complex cellular level. This complexity raises issues in itself due simply to the high compute requirement. It therefore seems necessary to find alternative ways of solving these "multi-scale" problems.

Using examples from the bio-fluid/engineering area we have presented an algorithm which can be used to "link" two differing computer architectures together in what has become known as System Level Acceleration. This allows for a substantial increase in the complexity of the problem to be solved and also in the range of scales that often inhibits the scientist/engineer.

4. Acknowledgements

The author would like to gratefully acknowledge the expertise from the IBM team of Jim Sexton, Abbas Farzadel and Al Gara and to Murray Price and Wayne Goss. Very special thanks to Dr Steve Moore for invaluable help also to the Blue Fern team of Tony Dale and Colin McMurtrie.
References


Grid Computing for Financial Applications

Patrizia BERALDI\textsuperscript{a,2}, Lucio GRANDINETTI\textsuperscript{a}, Antonio VIOLI\textsuperscript{a}, Italo EPICOCO\textsuperscript{b}

\textsuperscript{a} Department of Electronics, Informatics and Systems, University of Calabria
\textsuperscript{b} University of Salento, Lecce, Italy

Abstract. The paper investigates the potential advantages related to the adoption of grid computing technology in the financial sector. The increasing power of computational grids makes now possible to offer functionalities that were previously unimaginable. From derivative pricing, to risk management, from on-line trading to portfolio optimization, the possibility to manage huge amounts of data, solve models of seemingly limitless complexity and perform near real-time analysis may give the business a competitive edge. This contribution presents an important attempt to bring grid computing to the financial sector. In particular, it presents a grid-based platform for portfolio optimization designed and implemented within the European funded BEinGRID project. A system demonstration from the end-user perspective together with architectural details on the grid platform are provided.

Keywords. Grid Computing, Financial Application, Portfolio Optimization

Introduction

Nowadays, Grid Computing is gaining an increasing popularity in the financial service industry. This interest is motivated by the peculiar features of the financial sector: high volatility and growing complexity. Market conditions vary continuously making future evolutions difficult to predict. Moreover, the globalization of economic activities and the continuous introduction of innovative financial products have produced an increase in the complexity level of financial planning operations. In this landscape, also surrounded by new regulatory and institutional constraints (i.e. the new framework imposed by the Committee of Basel\textsuperscript{3}) and spectacular breakdowns, financial service providers need effective and efficient systems to operate in a competitive way. To this aim, the review of the enterprise IT infrastructure becomes a key priority. A first alternative is represented by the use of (massively) parallel supercomputers (see [1], [2], [3]). Their use, often well grounded in academic environments as proved by different funded research projects as, for example, AURORA ([4]), has met limited application in real-life operative contexts. This is mainly due to the high investment cost and the burden for administration and maintenance. Grid computing offers a more valuable alternative, especially for large fi-

\textsuperscript{1}This work has been partially supported by EU FP6 IP BEinGRID Project
\textsuperscript{2}Corresponding Author: Department of Electronics, Informatics and Systems, University of Calabria, 87036 Rende, Italy; E-mail: beraldi@deis.unical.it.
\textsuperscript{3}http://www.bis.org/publ/bcbs107.htm
nancial institutions which usually underuse their computational infrastructure. By pooling computing resources within an organization or across organizations, it is now possible to increase computational power by an order of magnitude. This new technology allows the financial industry to offer services of higher level in shorter time. Vast amounts of data, gathered from heterogeneous sources, can be handled and analyzed. More factors can be incorporated and managed in risk modeling equations, thus producing more confident results and improving the accuracy in risk management. Processing time can be drastically reduced for all time-sensitive financial applications, as for example, online trading. For a volatile stock or bond, a trading decision based on a four minute old data may be no longer valid. Thus, every second that can be saved offers a potential step ahead of the competition.

In Section 1 we briefly summarize the main experiences presented in the scientific literature on the use of grid computing for financial applications. They mainly concern the implementation of Monte-Carlo simulation techniques for option pricing and portfolio risk management. Somehow related to the area of portfolio optimization is also the design of grid-based solution methods for stochastic programming problems which are widely used to model applications in economics and finance. This specific research activity has been mainly carried out within the METANEOS\textsuperscript{4} project whose aim is the design of metacomputing environments suitable for solving optimization problems of unprecedented size and complexity.

In spite of the real advantage related to the use of grid computing, their diffusion in the financial service industry is still limited. With the aim of bridging the gap between academic research and industry, some research project initiatives have been recently promoted. Among others, we mention the European funded BEinGRID (Business Experiments in Grid) project, whose goal is to evaluate the potential impact produced by the adoption of this technology in different application sectors.

In Section 2 we present the design of the grid-based platform for financial applications, its implementation, and experiments and results. Conclusions and future research directions are provided in Section 3.

1. Literature Overview

Many applicative problems arising in the financial field are strictly connected to the simulation of uncertain market conditions. This constitutes a very challenging and computationally demanding activity, that involves several crucial applications such as derivative pricing, portfolio risk management and credit risk assessment. The main critical issue consists in representing the evolution of relevant risk factors (such as interest rates, credit default probabilities, stock prices) by means of stochastic processes. The interesting variables, such as interest rates, credit default probabilities, stock prices, are usually modelled using stochastic differential equations systems, like the well known single-factors models, such as the Vasicek model, the Cox-Ingersoll-Ross model or the exponential Ornstein-Uhlenbeck process (see \cite{5} for a detailed description). When a direct analytic solution of the equations is not possible, the standard approach is to make use of Monte-Carlo simulation. It is a statistical method that produces a number of possible variates according to a specified probability distribution taking into account possible correlation

\textsuperscript{4}www-unix.mcs.anl.gov/metaneos
among variables. For example, in risk management Monte-Carlo simulation is usually employed to determine the Value at Risk (VaR) of a portfolio, that is the maximum expected loss over a given holding period, at a predefined level of confidence. In particular, the Monte-Carlo technique is used to simulate the evolution of the single asset, by taking also into account the correlation among the random variables by means of the Cholesky matrix of the correlations. Thus, the portfolio value evolution can be easily determined starting from the single asset weight. The results of the simulation provide a sample of the possible realizations of the uncertain parameters that can be used to perform classical statistical analysis. The higher the number of the simulation trajectories performed, the larger the size of the sample and the more accurate the results. The nature of Monte-Carlo simulation fits the grid computing paradigm perfectly: workload can be easily partitioned into independent tasks to be assigned among the available computing resources.

The implementation of Monte-Carlo simulation on grid environments is the subject of some recent contributions. In [6] the authors propose a Monte-Carlo grid for financial risk management. In particular, they analyze three key technological issues, i.e. data protection, integrity and deadline scheduling which are mandatory to build a secure PC-grid based application.

The use of grid computing for portfolio and investment risk management has been explored in [7]. In particular, the authors evaluate the performance improvement in terms of execution time reduction and better quality estimation for the VaR computation of a given portfolio.

In [8] Monte-Carlo simulation is used for GARCH-based option pricing. Results are provided for a FPGA-based supercomputer, called Maxwell, developed at the University of Edinburgh. In [9] the authors present a comparison of the performance of two distributed algorithms for Bermudian-American option pricing implemented on a master-worker grid framework.

The solution of huge optimization problems arising in finance (as, for example, the classical Asset-Liability Management problem) represents another fertile area for the application of grid computing technology. Here the main challenge is the solution of structured optimization problems whose huge size prevents the application of standard sequential solution software. Typically, portfolio optimization problems are modeled by means of the stochastic programming paradigm. With respect to the deterministic counterpart, the use of this modeling framework allows to jointly capture the main features (uncertainty and dynamism) characterizing real applications. In stochastic programming models uncertainty is represented by a scenario tree (eventually defined by Monte-Carlo simulation). The larger the size, the more accurate the uncertainty representation. Unfortunately, the size of the resulting optimization problem is a function of the number of scenarios. Thus, accurate and robust solutions for problems of practical interest can be obtained if adequate computing resources are available. These considerations have motivated the design and implementation of specialized solution methods for stochastic programming problems on computational grids. In [10], the authors propose a specialized decomposition method for two-stage stochastic programming problems. The computational workload related to the solution of the second stage problems can be carried out in parallel since we have a sub-problem for each scenario. Computational experiments have been carried out on a grid computing platform based on the CONDOR system. Even though results have been collected on test problems not related to the area of finance, the method could be in principle used to solve portfolio optimization problems.
2. A Grid-based Platform for Portfolio Optimization

In this section we present our experience on the use of grid computing for financial applications carried out within the BEinGRID project. In particular, we describe a grid-based decision support system designed to address a typical problem arising in the area of portfolio optimization. The system relies on the integration of simulation techniques used to determine scenarios of uncertain future market conditions and advanced optimization methodologies to formulate and solve the corresponding mathematical problems. Before introducing the specific application, we provide a high-level snapshot of the grid-based platform. The system has been designed in functional modules that can be potentially used separately. Computational results on system performance together with a demonstration of system use from an end-user perspective are provided.

2.1. Architectural issues

Figure 1 shows the overall architecture of the grid-based platform. It is based on a three-tier structure and has been designed to be easily accessed by web. In particular, the front-end is a user-friendly web-based interface that provides access to the portfolio optimization applications and high level tools for managing the underlying grid resources. The Grid Portal indeed provides different functionalities based on a per user role access policy. The Grid Administrator has the possibility to handle grid resources, monitor the status of the resources and handle the user accounts. The middle layer is represented by the high-level grid services that efficiently manage the load balancing and the grid resources. The main components are the GRelC data service ([11]) and GRB metascheduler ([12]) that orchestrates the underlying middleware services. The third layer includes computing nodes each with an application instance, a local resource manager and a data manager to handle historical financial data, simulation and optimization results.

It is worthwhile noting that in the design of the grid-based system, several critical issues have been taken into account. The first one concerns the “robustness” of the solutions. This aspect is somehow related to the possibility to process huge amounts of historical financial data which are often gathered from different sources. Here the challenge is related to the accessibility of heterogeneous data, spread over different data sources and sometimes stored with different metrics. The solution adopted for “Data Management” is based on building a data federation among the available data sources. For performance reasons data federation is made both for using a unique database and directly accessing remote data sources. The heterogeneity of data sources is addressed using GRelC services. The Data Manager is configured through an XML file with the list of the available data sources and their features. This mechanism can be feasible without affecting dynamicity because the list of the data sources does not change rapidly.

A second important issue concerns the usability of the entire system which has been designed to be easily accessible by a wide community of end-users without any expertise on computational science. The proposed solution provides a “Grid Portal” based on the GRB and includes: user credential management, user workspace, job submission, job status checking, file transfer services.

The management of the grid resources and information services represent other critical issues. As for the first one, the grid platform adopts the GRB metascheduler as heart of the system. Some of the outstanding features of the GRB include: support for structured jobs, automatic refresh of user credentials, management of multiple credentials as-
associated to the same job, management of resources belonging to heterogeneous infrastructures based on different middleware. At the moment GRB supports Globus Toolkit 4\textsuperscript{5} and gLite\textsuperscript{6} middleware. Different information services have been analyzed. The BDII database\textsuperscript{7} adopted within the gLite services and iGrid based on a relation approach have been taken into account.

Finally, we mention the “security” issue. The system adopts the GSI infrastructure\textsuperscript{8} for user authentication and different authorization mechanisms based on ACL and X509v3 certificates with VOMS extensions. Among the others, one of the key aspects is the support for multiple credentials associated with a user. In other words, the system allows the simultaneous use of all the resources the user has access to in the same submission.

2.2. The system kernel

The grid-based system has been designed to address a specific financial application suggested by the BEinGRID end-users partners (Monte dei Paschi di Siena and Bank FINNAT EuroAmerica) referring to a Strategic Asset Allocation problem (SAA, for short). This is a classical problem that every financial institution faces at strategic level.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1}
\caption{Overall architecture of the system.}
\end{figure}

\textsuperscript{5}http://www.globus.org/toolkit/
\textsuperscript{6}http://glite.web.cern.ch/glite/
\textsuperscript{7}https://twiki.cern.ch/twiki/bin/view/EGEE/BDII
\textsuperscript{8}http://www.globus.org/security/
Given a set of “asset classes”, i.e. macro-sets of financial instruments identified by a common reference index, the problem is to define a medium-long term financial plan guaranteeing a given return and to minimize the risk. The considered problem belongs to the more general class of Asset-Liability Management problems, which have been deeply investigated in the scientific literature ([13], [14], [15]). The main difficulty in dealing with the considered application is that it concentrates issues difficult to deal with even separately. In fact, the problem is dynamic since actions are taken along the entire planning period. It is intrinsically stochastic since the uncertain evolution of the market conditions should be explicitly addressed in order to provide meaningful recommendations. In addition, the problem is highly constrained, since a large number of regulatory, strategic and risk aversion constraints should be considered in order to mathematically represent the end-user concerns. The methodological choice adopted to address the SAA problem is based on multistage stochastic programming ([16]), a sophisticated framework recently used to address other interesting problems arising in the financial area ([17], [18]). In contrast to the classical deterministic static models as the keystone Markowitz model ([19]) which implements a buy-and-hold policy, stochastic programming models allow to explicitly account for the dynamic and uncertain nature of the problem. Given a time horizon, the problem is formulated in order to consider the possibility to revise the initial decision concerning the portfolio composition. As time progresses and new information about the market conditions become available, portfolio can be rebalanced by buy/sell decisions. Moreover, specific measures are included to account for risk management.

The proposed framework is based on the integration of simulation and optimization techniques. The uncertain future evolution of the market conditions is represented by means of a scenario tree and information related to single nodes provides input data for the mathematical model. If more scenarios provide a faithful representation of the uncertainty, their number, on the other side, effects problem size, and, thus, in turn the solution process, calling for an advanced computing platform.

The following figure 2 shows a high-level snapshot of the system kernel.

It has been designed in functional modules sharing a common database accessed by an API interface. More specifically, stored data refers to historical information concerning prices of financial instruments, values of macroeconomic variables, generated scenarios and old portfolio compositions that the end-user may access at future time. A detailed description of the single modules together with the methodological choices are described below.

**Statistical Analysis Module:** analyzes and elaborates the huge amount of data storing historical information for deriving some statistical measures (mean, variance, covariance) which will serve as input for the next module. The module has been implemented in C++ and Matlab, and is linked to some advanced scientific libraries (i.e. GSL\(^9\)).

**Scenario Generation Module:** generates a scenario tree representing the uncertain future evolution of the market conditions by an econometric model. In particular, the model is based on the assumption that asset classes can be clustered in a limited number of sets around relevant market indexes, eventually correlated among each other, according to the results of statistical analysis. The market dynamic is strongly correlated with the number of real-economic variables and the evolution

---

\(^9\)http://www.gnu.org/software/gsl
of asset classes depends on the currency area they refer to. Before generating the simulated values, the scenario tree is created. Figure 3 shows a tree for a time horizon of four periods with a total of 27 nodes.
Given a time horizon and a user-defined time step, the tree generation is carried out level by level, specifying for each node a given number of children together with their probability of occurrence. For each node, the possible evolution of economic indexes for each currency area is simulated by means of a Monte-Carlo procedure based on an economic and capital market model and then returns and prices of the individual asset classes are calculated using a single-factor model, whose parameters are estimated on the basis of historical data. The module has been developed in C++ and is linked to some financial libraries (for example, Financial Numerical Recipes\textsuperscript{10}) to perform specific calculations. Additional details on the scenario generation techniques can be found in ([20]).

**Solution Kernel:** the generated scenarios represent the input data of the optimization model aimed at defining the optimal dynamic financial plan. The proposed model, whose details can be found in ([21]), incorporates specific restrictions on portfolio diversifications defined by the end user and has a risk-reward structure. In partic-

\textsuperscript{10}http://finance.bi.no/bernt/gcc_prog/recipes/index.html
ular, the objective function is represented by a weighted sum (by means of parameter $\lambda$) of the expected final wealth and the Conditional Value at Risk (CVaR) as risk measure. The mathematical problem is generated by means of a C++ module which extends an open-source modern algebraic modelling language, FLOPC++ ([22]) of the COIN-OR project, which can be interfaced with many state-of-art solution solvers. In particular, the CLP\footnote{https://projects.coin-or.org/Clp} (also in the COIN-OR project) serves the purpose of the solution package.

**Solutions Analysis Module:** the recommendations of the model are presented in figures and tables and are deeply analyzed by the solution analysis module. In particular, this module performs a scenario based portfolio analysis, with evidence of the wealth evolution in the best and worst cases. Moreover, for each previously defined solution it allows to calculate some statistical properties on the portfolio and the values of relevant risk measures. The robustness of suggested financial planning is also evaluated by means of a sensitivity analysis. Finally, the solution is compared with other well-known decision approaches for portfolio management (static Markowitz model, fix-mix, etc.).

### 2.3. Computational Experience

The system has been implemented on the Southern Partnership for Advanced Computational Infrastructure (SPACI\footnote{www.spaci.it}) grid, based on Globus middleware and made of about 450 working nodes. The testbed is composed as follow:

* a single processor node - resource A
* a dual processor node - resource B
* a farm with 5 dual processor nodes - resource C
* a parallel cluster with 60 dual processor nodes - resource D
The computational resources are networked together through a 100Mbit/s switch and connected to the external world through the GARR infrastructure. Each computational resource is equipped with different services/software (i.e. OS: Linux, Apache Web Server, GT4 libraries). Details can be found in [23]. More specifically, resource A is used to host the GRB Portal which represents the entry point for the grid infrastructure. The experiment considers two different kinds of resources: resources B and C to demonstrate the submission on a gLite-based infrastructure; and resource D to demonstrate the submission onto a GT4-based machine. These resources are also used to run the parallel version of the application.

The system has been designed so that Grid is “transparent” to the financial operator who can easily access the platform by means of a login interface (see fig. 4) and can select the application he wants to use (fig. 5).

Once the specific parameters are set (fig. 6), the request can be submitted and its status can be monitored (fig. 7).

Once successfully completed, the solutions can be visualized. Different analyses can be performed. First one, the efficient frontier (fig. 8), i.e. the set of non dominated solutions obtained for different values of the risk aversion, is visualized. This information is very useful for a financial operator since it provides the tool for evaluating a return/risk trade-off.

Once the best solution according to the risk preference is selected, (i.e. for a specific value of \( \lambda \)) the end-user can access relevant statistics on the proposed financial planning, from the best-worst case analysis to the wealth distribution over the set of scenarios (fig. 9). Finally, the end user can explore the proposed asset allocation along the planning horizon for each path of the scenario tree, together with wealth (and loss) evolution.
Figure 5. Application selection

Figure 6. Request set-up
Figure 7. Request status check

Figure 8. Efficient Frontier
Such a detailed level of information, obtained through the elaboration of a huge amount of data otherwise impossible to carry out, represents a real added value for the financial operator who can capitalize his experience by means of the advanced system.

Besides the user-friendliness, the grid-based system should be evaluated by measuring its computational performance. The design of the grid system allows more end-users to simultaneously access the portal, submitting the requests and obtaining the results. For a specific request, the computational advantage related to the use of the grid platform has been evaluated by measuring the scalability of the parallel version of the two more computationally demanding modules, the scenario generator and the solution kernel. The first one has been parallelized considering a balanced partition of nodes of the scenario tree among the available processors. This approach has been facilitated by the designed Monte-Carlo procedure which doesn’t rely on the father-children relation among the nodes of the tree. In fact, each processor computes assets returns for a subset of nodes and then sends the information to a master that determines (by means of a log transformation) the corresponding prices.

For the solution module, an “embarrassingly” parallel distribution of the computational workload has been considered. Each parallel task consists of the solution of the multi-stage model for a given value of the risk aversion parameter ($\lambda$). For a given $\lambda$, higher saving in the solution time can be obtained for larger size problems (larger number of scenarios, longer time horizon, higher cardinality of the universe of asset classes). Figures 11-12 and 13-14 show execution times and speed-up for different numbers of processors used.

The figures clearly show the performance improvement in function of the number of processors. It is worthwhile noting that further improvements can be obtained by paral-
Figure 10. Scenario analysis

Figure 11. Execution time of Scenario Generator
Figure 12. Speed-Up of Scenario Generator

Figure 13. Execution time of Solution Module
Lelizing the solution of the mathematical problems. This aspect has not been addressed in the current version of the grid-based system and it is the subject of ongoing research.

3. Conclusions

Today grid technology offers a computing infrastructure of increasing power enabling functionalities that were previously unimaginable. This advantage becomes particularly important in a sector like the financial one where applications are sensitive to time and space. From derivative pricing to risk management, from on-line trading to portfolio optimization, the possibility to manage huge amounts of data, solve models of seemingly limitless complexity and perform near real-time analysis may have a major impact for the financial service industry.

Moving to grid computing represents a cultural shift for an organization. For this reason in spite of their wide acceptance in academic environment, their application in commercial production environments is still limited. With the aim of bridging the gap between academic research and industry, some project initiatives have been recently promoted. This contribution witnesses the potential impact deriving from the adoption of the grid computing technology in the area of portfolio optimization. To make the grid computing technology pervasive, particular attention should be devoted to making it “transparent” to end-users that should access the grid platform without being aware of its complexity. In this respect, the system designed and implemented within the BEinGRID project represents an important step forward.
References


This page intentionally left blank
Chapter 6

HPC and GRID Infrastructures for e-Science
AN ACTIVE DATA MODEL

Tim Ho and David Abramson
Monash e-Science and Grid Engineering Lab
Faculty of Information Technology
Monash University

Abstract. The Grid allows scientists to design and perform very large-scale experiments that are not possible previously. Many of these experiments produce a large number of data sets and significantly the amount of data being captured, generated, replicated and archived is growing at an astonishing rate. We have designed and prototyped an Active Data System that provides a complete, automated solution to problems that arise in the management and curation of derived data. Significantly, the system allows users to recompute data rather than necessarily store it, and adds a layer that provides efficient access to remotely distributed replicated sources across different middleware stacks.

Keywords. Data replication, workflows, distributed file I/O, virtual data, curation, data lifecycle

Introduction

The 21st century has been described as the century of data. Various groups predict an exponential growth in the amount of data that will be captured, generated and archived [20]; some researchers predict a tenfold increase of data every five years from 2000 to 2015 alone [10]. Managing this growth will be challenging, and will require significant effort in the areas of data management and curation. Not surprisingly, there are numerous projects addressing the underlying computer science challenges, and a number of specific middleware solutions are being developed.

This chapter concerns problems that arise in the management and curation of derived data; data which is produced (and consumed) by large computational models, as opposed to data that is captured from real world experiments. In particular, we develop tools and techniques that allow us to recompute data rather than necessarily store it, thereby reducing the impact of derived data on storage systems. Furthermore, our overall approach builds on existing data replication services in a flexible manner, and adds a layer that provides efficient access to remotely distributed sources across different middleware stacks.

1. Motivation

Grid applications can generate massive amounts of data that is stored, replicated and distributed across multiple sites. For example, the digital sky project [36] processes
digital images of the sky and involves sharing and analysis of terabytes of pixel data (stored in millions of files) across multiple institutions around the globe. Some experiments (such as [19]) also generate large volumes of derived data products (from large data pipelines) that need to be stored, either temporarily or permanently, for further processing and analysis by other groups that are geographically dispersed.

Several replica systems have been developed to assist the management of data on the Grid [4][7][25]. In general, these systems provide mechanisms that allow users to discover, replicate, register and access data that may be stored in multiple resources, such as network file systems and mass storage devices. Importantly, they provide means for the users to keep track of the data copies that may be created and accessed by individuals and groups from one or more virtual organizations.

Furthermore, there is a well recognized need for the curation and preservation of research data [16][27][28][29]. A curated database not only stores archived data, but also contains descriptive information about the datasets, such as how and why the data was produced, and the relationship between them. Using this extra information, users other than the dataset’s creators may discover, reuse, reproduce and validate the data, avoid duplicating existing work, and develop new research based on it. For example, a user may want to find out what parameters were used in an experiment conducted by other groups, and then fine-tune the result by adjusting the parameters.

The motivation for the research described in this chapter can be best illustrated by the following scenario:

John executes a simulation that generates a large number of derived data products which in turn feed a complex processing pipeline across multiple distributed sites.

In order to provide efficient local access for the downstream processes, he copies the data to multiple geographically dispersed locations. John collaborates with many different partners, but there is no agreed standard way of storing and maintaining the data sets. In fact, each partner site insists that they use their own data management and replication middleware.

A few months later, after the experiments are completed, John is asked to release the disk space. He decides to delete the output of his simulation, including all distributed copies. It is a long and tedious process, complicated by the lack of a single data management system.

A year later, a group of researchers develop a new simulation model based on John’s work. They want to analyse the original derived data and explore possible improvements and optimization. However, all data has been deleted, and John cannot remember how to recreate it. Worse, even if they found a copy, it wouldn’t be very useful because John has never described how the data was produced.

This example shows a number of issues that researchers are facing with the deluge of derived data; we discuss these issues below.
Scale of data

**Impact on storage systems**
Computational models may generate large amounts of data. In general, the data requires active management so that it can be discovered and reused by others. While data replication may improve performance and fault tolerance, it actually makes the storage problem worse because there are multiple copies.

There is a need to balance the costs, benefits and the anticipated needs of future generations when considering storing data indefinitely; this is a long-term commitment and requires long-term funding. Clearly, at some point, the cost of maintaining large archives of derived data products needs to be justified.

**Store versus delete**
Much data captured by instruments (such as satellites and telescopes) is non-reproducible. Such data, often referred to as observational data, needs to be stored because there is no way to recreate it.

However, derived data created by computations is often reproducible. For example, simulation models may produce large amount of derived data, which can be reproduced by running the original model. Of course, this assumes that we have sufficient information about how the model may be run. Such information can be added to the data in the form of metadata. In the long run, it may be cheaper to store the model and the metadata and then recompute the data when requested.

**Grid data access**

**Heterogeneity in the replica systems**
While a number of replica systems are available [4][7][25], most of them employ different data access interfaces that are not compatible with each other. Thus, code that is built on one system cannot easily access data managed by another. This drawback is significant; an application may require access to datasets shared by multiple virtual organizations, with each of them using different data management middleware.

**Replica selection in a dynamic environment**
Conventionally, when multiple replicas exist, users must choose one manually. A replica that delivers the lowest possible response time would provide the best data access performance among the replicas. However, when many replicas and metrics (for the selection criteria) are involved, it becomes a tedious task.

While much effort has been spent on automated selection algorithms [6][34][37][42][44], they typically only select one single replica for each request and assume that it will be accessible during data transfers. We believe this is not adequate because the Grid is dynamic; shared resources are changing in this environment.

2. Active Data Model

**2.1. Grid Data Life Cycle**

In general, applications can be characterised as lying on a spectrum that ranges from being totally I/O bound to totally compute bound. The former refers to applications that
spend significantly more time on data access operations than processing. Thus, their execution time largely depends on the rate at which data can be obtained. In contrast, compute bound applications perform significantly more processing than file I/Os and thus the execution time mainly depends on the CPU speed, the amount of main memory and the algorithm used within the application code.

Clearly, fast execution performance for an I/O bound program can be achieved by running it as close as possible to the required data. In fact, running the program on the machine where the data resides is the only way to achieve the best possible execution performance because no remote file operations are required. On the other hand, the best execution performance for a compute bound program can be achieved by running it on a machine that provides the fastest hardware in a Grid.

However, the vast majority of applications sit somewhere between completely I/O bound or compute bound. For example, the Large Hadron Collider (LHC) experiments [26] require access to very large datasets distributed worldwide for series of compute-intensive activities. Thus, the performance of these experiments not only depends on the performance of the hardware, but also on how fast they can obtain the required data. Remote access performance can be improved by copying the required data to a location that is close to the clients. In fact, one may prefer to maintain a replica locally, particularly when it is frequently used, because this avoids remote access overheads entirely. Importantly, this is not always possible; a dataset may exceed the storage capacity provided by any single machine; in which case, it is split into multiple portions that are stored on different resources.

Data replication also improves scalability because it allows users to distribute identical copies to multiple sites, rather than storing everything at one location. Having replicas spread across multiple machines also improves application reliability and reduces the risk of losing data because the data will continue to be available even in the event of hardware failure.

We have identified a data lifecycle that models the whole replication process; significantly, we want to use it to provide automated solutions to the following issues, as introduced in Section 1:

**Scale of data**

The amount of derived data on the Grid is growing at an astonishing rate. This places significant impact on storage systems (both hardware and software). A data lifecycle addresses this issue by allowing users to completely delete reproducible data, such as derived data products that are produced and consumed by computations, and concerns mechanisms for regenerating the data transparently.

**Grid data access**

The Grid Data Life Cycle (GDLC) models the replication process and concerns mechanisms that provide access to remote replicas. We use existing replica systems for data management, and provide an abstraction layer above their interfaces. This layer allows applications to source data from different systems seamlessly. Furthermore, when multiple replicas exist, computations may not only acquire data from one single source; we provide support for dynamic source selection, in which the best replica will be chosen dynamically in runtime based on real-time metrics, such as the bandwidth
and latency between the client and the replica servers. In particular, data source can be changed at runtime when a better one (e.g. faster) is available.

The lifecycle consists of a number of states. When a computation executes, it generates some output data that may later become input data of some other models. The data may be mined, analysed and refined to produce other datasets. It may also be replicated to different locations for further processing. Later on when the data and its copies are no longer needed, they may be deleted. In the end, this would result in complete data removal.

However, deletion is not the final phase of this lifecycle. Building on the idea of virtual data [14], we view data regeneration as a special case of replication; data does not need to be physically stored, but instead exists as specifications that describe how it may be produced (or reproduced). When a program reads from a file that was produced by a computation, but the data has been deleted, then the file could be regenerated by rerunning the computation.

We call this the Grid Data Life Cycle (GDLC), as shown in Figure 1; from no data, to one copy of data (computation), to many copies of data (replication), to no data (deletion), and back to one copy of data (regeneration).

2. Active Data: Supporting the GDLC

Based on the GDLC, we have developed an Active Data (AD) model [24][21], which provides a seamless pathway to data stored in heterogeneous systems and, significantly, enables access to virtual files, in which derived data can be dynamically created. Specifically, the model concerns a number of requirements for supporting the processes (including computation, replication, deletion and regeneration) within the lifecycle.

2.2.1. Accessing data from dynamic, heterogeneous resources

The GDLC starts when data is produced by a computation, which changes the state of the lifecycle from no data to one copy of data. This data is typically derived from other data; a computation reads from a primary dataset (which may be generated by an instrument or other computational models) and then produces one or more data products that may be used by other models in a pipeline. Figure 2 shows a sample workflow application that creates some derived data products.
Conventionally, the location of the data is fixed; for example, the path to a local file is specified in the open system call. This also applies to programs that use file paths set by command line arguments or environment variables because the paths typically do not change when the programs are running. This is not adequate for a Grid application because shared resources may be changing during runtime in a wide area environment; one should not assume that a data source will be always accessible.

When a dataset is not available locally, users may copy the data from a remote storage server, or move the computations to a host where a replica is available. In fact, in many cases, a combination of both is required when an application and the required data collection spans multiple sites. Moving data across networks can be very inefficient, particularly when only a small subset of a large file is required. Moving computations may be infeasible if they require specific hardware configurations and/or software libraries that are not available on the host.

Therefore, it is desirable to use non-fixed file paths in the code, which is commonly referred to as logical file names; a logical file may be mapped to any data source, such as a local file or a file stored in a remote system. Moreover, the mapping is changeable and can be dynamic. This is flexible because users can choose and change the sources without modifying the application code; only the file mappings need to be changed. This may also be done dynamically by a job submission system.

Furthermore, as illustrated in Figure 3, an application may span multiple organizations and require access to data collections stored in multiple replica systems. This is problematic because these systems often employ interfaces that are not compatible with each other. Consider a project that has distributed terabytes of experimental results using the Storage Resource Broker [5]. Later, the project is involved in a collaboration based on Globus Replica Location Service (RLS) [8]. In order to share and analyse the data, the users of both projects have several choices. They may agree on using one replica system and move all data to it, or they can employ third-party file copy tools, and/or modify the applications to access data from both systems. Moving data may be expensive and can lead to incoherence problems when the data is updated.

Moreover, a program may only require access to small portions of a large file, in which case copying the entire file can be inefficient. On the other hand, code modification tends to be error prone and is sometimes infeasible when the source code
is not available, such as commercial programs. In fact, some programs are simply too fragile to be modified. This also means that developers of one system must have knowledge of the other, and are required to learn new interfaces and write specific code for each of the systems. In addition, since the application is built on specific interfaces, the code needs to be changed in order to support other systems; such as when a data collection is moved.

Clearly, a data access layer, as shown in Figure 4, which separates the application from the underlying systems (such as [5][8][38]) would offer great flexibility in supporting dynamic sources. Specifically, this layer provides an interface that abstracts various data access mechanisms and hides from the applications the complexity of accessing data from multiple middleware stacks. It should also be extensible, so that a wide range of mechanisms can be supported without any change to the layer and high level services.

![Figure 3: Accessing data from virtual organizations](image)

![Figure 4: A data access layer](image)

Furthermore, developers only need to learn one interface, and do not need to know what data sources to use and how they are accessed at design time; these are done within the data access layer, so that applications are able to adapt to multiple heterogeneous systems during runtime without code modification.

The support for dynamic sources is also useful for scheduling and running jobs in a Grid. Depending on user preference, a scheduler may run a program on a resource that is close to the required data, or on a resource that offers the best computational performance. The latter may require the program to decide which server to use when multiple replicas exist; a decision is made at runtime, taking consideration of what sources are available and their real-time condition (such as the network latency).
Moreover, a scheduler may instruct a program to write data to a remote location where a computation that requires access to the data is running. This is particularly efficient for parallel processing because the data can be immediately made available and thus, downstream processes do not need to wait until all driving computations are finished.

### 2.2.2. Accessing data from multiple data sources

The GDLC models the entire replication process; from no data, to one single copy, to multiple copies of data. Our model does not concern mechanisms for replica management, such as user authentication, access policy and data consistency; these are already supported by existing middleware systems, such as the Storage Resource Broker (SRB) [5]. However, we do want to provide seamless and transparent access to data stored in these systems, so that computations are able to adapt to a wide range of systems dynamically.

Data replication helps reduce access overheads by bringing data closer to clients, and improves scalability and reliability by having data copies in multiple locations. When there is only one data copy, and a replica is created, then the state of the lifecycle changes from one copy to multiple copies; subsequent replication does not change the state.

When multiple data sources are available, a replica selection service is required; it discovers the physical locations of the available replicas and then selects the best one based on performance criteria, such as the network performance between a client and the servers.

Importantly, replica selection should be made dynamically because a better source may become available, even during program execution. In the AD model, the system maintains a list of available data copies at runtime, and continuously monitors the condition of the resources, such as the latency, bandwidth and server load. When a replica is not accessible, or a better one is found, the system can switch to the better source transparently as soon as possible in order to avoid timeouts or prolonged program execution.

Significantly, dynamic replica selection realizes optimized data access; an application is able to source data from a server that offers the best access performance. Thus, data can be obtained in a fastest possible manner. Experiments show that dynamic replica selection is very efficient [22][23].

### 2.2.3. Deleting data from heterogeneous systems

Data deletion is an important process in the GDLC. In general, a user may delete data only when there is a replica available or there is a way to reproduce the data when needed. The complete deletion process is illustrated by a flowchart shown in Figure 5. When multiple data copies exist, users may delete any copy; this is because applications that require access to the data will still run normally as long as there is at least one copy. The state of the lifecycle changes from multiple copies to one copy when there is only one copy left, or changes from one copy to no copy when all copies are deleted. Importantly, to avoid data loss the last copy must not be deleted unless data regeneration information is created; we need this information in order to reproduce the data.
Figure 5: Data deletion flowchart

2.2.4. Storing regeneration information as metadata

Our model extends the idea of virtual data to support *data regeneration metadata*, which contains regeneration information that describes how to reproduce a *target file*. The information may include the location of an arbitrary program, details of a workflow, where computations should be run, what parameters to use, and locations of the input and output files. Moreover, it is associated with a target file, as illustrated in Figure 6; when this file is deleted, the metadata should be kept and maintained, so that we can reproduce the data later based on the regeneration information.

![Figure 6: Target file and data regeneration metadata](image)

The metadata may be specified by users manually. However, this can be error prone and difficult when many tasks (such as multiple computations and data transfers in a workflow) are involved. It may be better to automatically track and record what tasks and events are required to create a target file. This is advantageous because it requires little manual inputs and can provide accurate computational information. The Kepler provenance framework is a sample system that allows users to record what software components and data flows are needed.

The data regeneration metadata may be stored in a replica system where the target files reside. This is feasible because many replica systems, including SRB and Globus RLS, support *user-level metadata*, which allows information to be stored in a user-defined format. The metadata facilitates the use of *virtual file*, which contains data that does not physically exist, but is associated with metadata that describes how it may be produced. Some replica systems, such as Gfarm [15], do not allow user-level metadata.
In this case, a metadata database is needed, which records and associates regeneration information for virtual files. In fact, it may be possible to develop such service on top of other metadata systems, such as SRB’s MCAT [5].

Virtual files allow users to completely delete computed data (thus, reducing the amount of physically stored data), while maintaining details of how to reproduce it. Furthermore, users are able to discover data products based on the regeneration information. For example, one may discover data that is computed by a particular model and validate the quality of data based on what input data and parameters were used to create it. This is significant because it is often not possible to locate such information even in a very carefully curated database.

2.2.5. Accessing virtual files: on-demand data regeneration

In the AD model, users may delete a target file (and all its replicas), as long as it can be reproduced; we assume that data can be reproduced when regeneration metadata is available. The state of the lifecycle changes from no copy to one copy when data is reproduced by running one or more computations.

On-demand data regeneration is a process of producing derived data for a target file. When an application opens this file, but the data has been deleted, we can recompute the data by using the information specified in the metadata. Clearly, this requires tools that allow users to create and store the metadata, and a set of mechanisms for running the required computations and dynamically mapping virtual files to the physical locations of the computed data.

![File open flowchart](image)

Figure 7: File open flowchart

Figure 7 shows a flowchart that depicts the file open process. When a computation accesses a logical file that is mapped to a virtual file, data regeneration is required. Clearly, a file can be produced only when regeneration metadata is available, and the information in the metadata is valid; for example, all driving computations and data are available.

Furthermore, derived data may be reproduced by different data regeneration mechanisms, such as by running an arbitrary program or a workflow. In addition, software components may need to run on particular remote resources; for example, a computation may only work on a specific platform. In this case, the regenerated data
may be created remotely; we need to know where it is located and how it can be accessed.

Data regeneration should be done automatically and transparently. When an application opens a virtual file, we can block the execution, regenerate the data and then resume the process. Once the data is computed, it may be accessed by remote file I/O or staging; in the latter case, the application waits until all required data is computed and transferred to a local cache. Moreover, it may be possible to overlap the processes when the computations are based on a producer-consumer model. In this case, data produced during a regeneration process can be sent to a consumer immediately; this can significantly improve the performance, particularly when many processes are involved.

2.2.6. Support for legacy code

Unlike many other middleware systems that focus on the development of new Grid-aware applications, we are particularly interested in supporting legacy code. Specifically, we want to reuse the large number of software components that already exist for building new Grid workflows. In most cases, these components are written in a range of legacy languages, such as C and Fortran, and use conventional file I/O for local data access. Job submission systems, such as Nimrod, allow users to run existing programs on Grid resources without modifying their code. However, since these programs only read and write local files, data needs to be copied before and after execution; this can be costly and very inefficient when many files are transferred across networks.

Our model concerns flexible file I/O mechanisms that allow existing software to read and write logical files, which may be mapped to local files, remote files managed by various replica systems or remote buffers for data streaming. Significantly, this is done transparently based on GriddLeS [22][23], so that users are not required to modify and recompile the existing code.

2.2.7. Active Data versus other systems

The Active Data model has significant advantages to application developers because it provides an integrated solution to supporting the GDLC. To date, no other system provides similar functionality that addresses all requirements of the lifecycle in the areas of computation, replication, deletion, regeneration and legacy support. Here we provide a brief comparison to other systems; a summary is available in Table 1.

Remote data access plays a significant role on the Grid because shared resources, including computational units and data archives, are often distributed across networks. As a result, many projects are geared toward the development of middleware that provides high performance data transfer between clients and servers. An example of such middleware is GridFTP [1], which has emerged as a de facto standard for moving data in wide area environments. However, it is a rather low-level service and is not able to adapt to dynamic changes in wide area networks; the latter is typically supported by higher level services, such as Globus RFT [30] and DRS [9].
Furthermore, few data access middleware solutions address heterogeneity in replica systems. In particular, none of them supports dynamic, changeable data source, which allows applications to switch to a better replica automatically and transparently based on performance criteria. Moreover, most of them are designed for the development of new applications and do not provide support for existing software.

Significantly, almost no other systems concern data deletion and regeneration, in which computations may access virtual data that is not stored physically. One example of such system is VDS [14], (formerly known as Chimera). It stores data regeneration information in a virtual data catalog and uses definition statements to describe how derived data is created. From this information, the system is able to dynamically construct workflows that can be run on the Grid. VDS, together with Chiron [14], offers a job submission system that allows users to run these workflows that derive data from other data; when a required dataset does not exist, VDS can transparently create it when sufficient regeneration information is available. In addition, when workflow processes span multiple hosts, remote data is copied from one stage to the next.

Our model is very different from VDS. In particular, we focus on a data access layer (for application development) that sits above existing middleware stacks. This layer supports multiple data access protocols and is extensible without code change. It also provides optimized data access based on dynamic replica selection. Moreover, we consider different data regeneration mechanisms that support both blocked and non-blocked operations. In addition, we concern mechanisms that support the special data deletion process defined by the GDLC.

Clearly, existing middleware only provides piecemeal solutions to individual requirements for supporting the GDLC; no existing work offers a complete package that addresses these requirements in an integrated manner.

<table>
<thead>
<tr>
<th>Computation</th>
<th>XIO</th>
<th>RIO</th>
<th>Legion</th>
<th>SRM</th>
<th>GFAL</th>
<th>GAT</th>
<th>Bypass</th>
<th>Parrot</th>
<th>OGSA</th>
<th>VDS</th>
<th>ADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Support multiple protocols</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Extensible without code change</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Replication</th>
<th>XIO</th>
<th>RIO</th>
<th>Legion</th>
<th>SRM</th>
<th>GFAL</th>
<th>GAT</th>
<th>Bypass</th>
<th>Parrot</th>
<th>OGSA</th>
<th>VDS</th>
<th>ADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Support multiple replica systems</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Replica selection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Dynamic, changeable data source</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deletion</th>
<th>XIO</th>
<th>RIO</th>
<th>Legion</th>
<th>SRM</th>
<th>GFAL</th>
<th>GAT</th>
<th>Bypass</th>
<th>Parrot</th>
<th>OGSA</th>
<th>VDS</th>
<th>ADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Special data deletion mechanism</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Stores regeneration information</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regeneration</th>
<th>XIO</th>
<th>RIO</th>
<th>Legion</th>
<th>SRM</th>
<th>GFAL</th>
<th>GAT</th>
<th>Bypass</th>
<th>Parrot</th>
<th>OGSA</th>
<th>VDS</th>
<th>ADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Blocked regeneration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Non-blocked regeneration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Legacy support</th>
<th>XIO</th>
<th>RIO</th>
<th>Legion</th>
<th>SRM</th>
<th>GFAL</th>
<th>GAT</th>
<th>Bypass</th>
<th>Parrot</th>
<th>OGSA</th>
<th>VDS</th>
<th>ADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Build workflow from existing code</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>- Run programs without code change</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Feature comparison
3. Design of the Active Data System

3.1. Architecture

Our Active Data System (ADS) [24][21] consists of three major parts, namely, Active Data engine, Active Data plug-ins and Active Data client commands. Broadly, it supports various data access mechanisms using the Active Data engine, which provides an engine interface that is independent of the data access protocols and replica systems. This interface is a data access layer (discussed in Section 2.2.1) that uses existing Grid services, such as GridFTP, SRB and NWS [43], for remote data access, resource monitoring, metadata management and data regeneration. Access to these services is implemented as a number of Active Data plug-ins, which are lightweight modular software components used by the engine. Furthermore, a number of client commands are needed; these include a metadata command for managing data regeneration information, a file removal command that implements the special deletion process, and a runtime command for running existing software on top of the engine.

As shown in Figure 8, the ADS employs a modular and layered software architecture, which allows loose coupling of various software components. It consists of four different layers, namely, an application layer, an Active Data layer, a middleware layer, and a fabric layer.

**Application layer:** This layer refers to applications that are built on the Active Data engine. These include both new and existing programs, and the Active Data commands. Users may also develop high-level Data Grid services based on the ADS.

**Active Data layer:** This layer refers to the ADS. Specifically, the layer is represented by the engine interface used by the application layer. It allows applications built on the Active Data engine to access multiple Grid middleware systems using the one single interface. These applications request particular operations by calling...
the functions defined by the engine, which in turn passes the requests to corresponding plug-ins.

**Middleware layer:** This layer refers to the functionality provided by the Grid middleware systems. For example, a user may use SRB for replica management and NWS for resource monitoring.

**Fabric layer:** This layer refers to the physical resources available in a Grid, such as high-performance computers, networks, databases and instruments.

### 3.2. Active Data Plug-ins

The ADS employs a plug-in model for supporting a range of operations, including remote data access, resource monitoring, metadata access and data regeneration. We explore the various types of plug-ins for these operations in the following subsections.

#### 3.2.1. Data Access Plug-ins

The most common forms of data access are file transfer and remote file I/O. The former is typically supported by high-performance data transfer protocols, such as GridFTP, while the latter may be supported by various file I/O middleware, such as XIO and GFAL. Users may prefer one over another or a combination of both depending on the application (and performance) requirements. In fact, in many cases, the choice will be affected by what middleware a virtual organization provides.

Access to remote data is supported by two types of data access plug-ins, namely, file transfer plug-in and file I/O plug-in. File transfer plug-ins concern transferring files from remote sites to a local cache. In general, these plug-ins are built on top of existing file access protocols, such as FTP and GridFTP.

However, most applications access data in the form of file I/O; some may use conventional system calls for local file access, while some may access remote data using remote file I/O systems. These are supported by the file I/O plug-ins, which implement file access functionalities specific to particular I/O services, such as Globus GASS and XIO.

With the plug-in model, users of our ADS are able to configure various data access models. For example, when an application built on the engine interface requires access to a file from a remote machine, at runtime the file can be transferred from the remote machine using the GridFTP protocol and file operations are then performed locally using conventional system calls. This is similar to GASS, in which files are fetched to a local cache for all subsequent file operations. The copied file may also be registered as a replica, so that others may discover and use it.

#### 3.2.2. Resource Monitoring Plug-ins

Dynamic data source selection requires the resource information provided by a resource monitor, which uses existing Grid monitoring middleware based on the plug-in model (such as NWS). Each resource monitoring plug-in provides access to information created by one or more middleware and may predict future resource conditions based on the measurements previously recorded.

In general, the information provided (or forecasted) by a resource monitoring plug-in is used for choosing a replica server when multiple copies are available. Once selected, the server will be used for all subsequent file operations unless a better server
is found; when a better data source is found, the system will automatically and transparently switch to the new source.

3.2.3. Metadata Access Plug-ins

The ADS has a metadata store that provides means for users to store data regeneration information. Access to the information is supported by one or more metadata access plug-ins.

Data regeneration metadata may be stored in a replica system that supports user-level metadata. Therefore, metadata access may be implemented in data access plug-ins because in most cases both data and metadata access are supported by the interfaces provided by the replica systems, such as SRB and RLS. When no metadata access is provided by a data access plug-in, the metadata store will use its own database for storing the metadata and association.

Furthermore, it may be possible to build a data access plug-in on other plug-ins. For example, as shown in Figure 9, a Gfarm plug-in may support user-level metadata by using an SRB plug-in. In this case, access to Gfarm files is supported by the Gfarm plug-in. Since Gfarm lacks support for user-level metadata, the plug-in utilizes the metadata features provided by the SRB plug-in. Thus, data regeneration information for the Gfarm files can be stored in SRB. Depending on the implementation, the metadata may be identified using logical file names or the Gfarm file URLs.

![Figure 9: User-level metadata support for Gfarm using SRB](image)

3.2.4. Data Regeneration Plug-ins

Data may be created by various data regeneration mechanisms, ranging from simple command line programs to complex Grid workflows involving large number of distributed resources. Each of these mechanisms is implemented by a data regeneration plug-in.

Data regeneration requires two components, namely, the metadata store and a data regenerator. The data regenerator reads regeneration information from the metadata store, and uses this information to regenerate data. For example, a file may be produced by an arbitrary Unix program; in this case, a plug-in that supports execution of Unix commands is required. It also needs to know which file created by the program should be used and where it is created, so that the file may be accessed using file staging or file I/O.
A sample data regeneration process is shown in Figure 10. When the engine interface tries to open a virtual file, it sends a file request to the data regenerator. The data regenerator then looks for the regeneration information associated with the file. In this example, the detail of a remote command is found. The file is recreated by using a remote command plug-in to run the program in the machine specified in the metadata. When the regeneration is done, the data regenerator notifies the interface about the location of the newly created file. Finally, the remote file is accessed using a remote file plug-in.

Clearly, in the example above, the file open operation is blocked because it needs to wait until the required data is produced and copied. One may also develop data regeneration plug-ins that support data streaming for asynchronous operations.

![Figure 10: A data regeneration process](image)

4. Implementation of an ADS prototype

A prototype based on our ADS model has been implemented. The code is written in C and can be run on Linux. Currently, the prototype supports several replica systems, including SRB, Gfarm and Globus RLS. It also implements multiple data and metadata access interfaces and supports two data regeneration mechanisms. In addition, there is also replica selection support based on metrics from NWS and user-defined simulation data.

4.1. A unified interface for file I/O

The framework implements a unified interface (hereafter, the interface) that abstracts the underlying Grid middleware services. Specifically, the interface is designed to support both file and metadata access. Applications that may use this interface include high-level Data Grid services that span multiple middleware systems and the Active Data client utilities.

For distributed data access, there is a single uniform file I/O API. This API provides a number of file I/O functions that represent common file operations often required by conventional programs. These functions include open, close, read, write, stat and seek, which have arguments and return values very similar to the POSIX interface. This allows the developers to easily adopt the interface without a steep learning curve. Table 2 shows a list of system calls, the corresponding ADS file I/O functions and the file I/O functions provided by the data access plug-ins.
The ultimate goal of the interface is to forward all file operation requests to the plug-ins specified by the users. However, the interface also performs a number of tasks before and after interacting with the plug-ins for some ADS-specific operations. During the file open event, typically a logical file name (a file identifier) is passed to the ad_open function. This function looks for the physical file using the given file identifier (defined in local or global file registries that are described in Section 4.2) and then searches and opens a plug-in module that is responsible for providing specific mechanisms to access the file; for example, an SRB plug-in would eventually call the srbObjOpen function while a Gfarm plug-in would call gfs_pio_open. The interface also supports data streaming via a buffer service that connects reader and writer code, as discussed in Section 4.3.

In addition, when a required file does not exist, the interface will try to recreate the data using a data generator, which is described in Section 4.4. Moreover, when multiple replicas are available, a list of replica locations is created and the best data source will be chosen at runtime. Importantly, a source can be changed transparently without affecting a read operation; see Section 4.5 for more details.

<table>
<thead>
<tr>
<th>Syscalls</th>
<th>Active Data I/O</th>
<th>Plug-in I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>ad_open</td>
<td>p_open</td>
</tr>
<tr>
<td>close</td>
<td>ad_close</td>
<td>p_close</td>
</tr>
<tr>
<td>read</td>
<td>ad_read</td>
<td>p_read</td>
</tr>
<tr>
<td>write</td>
<td>ad_write</td>
<td>p_write</td>
</tr>
<tr>
<td>stat</td>
<td>ad_stat</td>
<td>p_stat</td>
</tr>
<tr>
<td>fstat</td>
<td>ad_fstat</td>
<td>p_fstat</td>
</tr>
<tr>
<td>lseek</td>
<td>ad_lseek</td>
<td>p_lseek</td>
</tr>
</tbody>
</table>

Table 2: Unix system calls, Active Data file I/O and plug-in I/O

4.2. Local and global file registries

A file mapping is a key-value pair stored in a file registry. The key represents a logical file name used by the users (and applications), while the value contains information about the physical file associated with the logical file. Logical file spaces are composed by creating file mappings, which are accessed by the framework during file operations, such as ad_open and ad_stat.

Figure 11 shows an example of a file mapping. In this example, the logical file name is file, which is mapped to an SRB object, file2. The physical file information is formatted in colon-separated tokens. The first token is a plug-in identifier (in this case, srb), which tells the framework what data access plug-in should be used for accessing the file. The other tokens refer to information that is specific to the corresponding plug-in; in runtime, these tokens are parsed and consumed by the plug-in. The framework is not required to understand this information, and thus is able to work with any physical files supported by the plug-ins.

File mappings can be stored locally or globally. A local file registry refers to file mappings that are stored in a mapping file, which is only readable by a particular user or group. A global file registry is realized by publishing a list of file mappings for data shared among multiple organizations. This is done via a web service, called registry service. Differ from the local file registry, global file mappings are stored in a mapping file that is globally accessible within a virtual organization and can be read by querying the registry service using SOAP calls.
4.3. File-based inter-process communication

Many experiments require the execution of multiple software components; outputs from a computation are required by one or more downstream components in a workflow. Typically, these components are run sequentially and, in many cases, manually by the users. Moreover, they are sometimes run on different resources because of reasons, such as hardware and software dependency, security and performance requirements.

The prototype supports file-based inter-process communication, in which data written by a producer can be made available immediately to a consumer. This allows the programs to be connected in a way similar to using pipes. When sufficient data is available, the consumer is able to perform computations without waiting. This allows certain degree of process overlapping and can improve the execution performance.

This is done by creating a buffer between a writer and a reader, as illustrated in Figure 12. This buffer may be created in the machine that runs the writer program, the machine that runs the reader program, or in a different machine that is accessible by both the writer and the reader. Moreover, the buffer is stored as a file (called buffer file) registered in a global file registry, so that the buffer can be accessed by programs running in different hosts.

The file buffer is managed by a web service, called buffer service. This service provides an interface for the framework to create buffer files and access data (read and write) in the buffers. Specifically, the framework accesses the file buffer using a remote buffer plug-in, which is a client of the buffer service and provides file I/O-based access to the buffer files using SOAP calls.

4.4. On-demand data regeneration

Our ADS prototype provides access to virtual files, which are files that are associated with data regeneration metadata. The support for this is twofold; the prototype provides
a client command for users to record data regeneration metadata and a data regeneration interface (used by a data regenerator) is provided for creating the data based on the metadata. Note that write operation are not supported because any change to the regenerated data will not change the way it will be reproduced in the future.

Currently, the prototype provides two data regeneration plug-ins; one supports the execution of binary programs and the other one supports the execution of a Kepler workflow. When a file is being opened for read, the interface first makes sure that the file is accessible. If the file is missing, the system will try to retrieve the data regeneration metadata from a metadata store. When sufficient data regeneration information is available (such as what plug-in to use for recreating a dataset, path to an executable binary file, command line arguments, etc.), the system will load the responsible plug-in module and pass all information to this plug-in, which executes the required programs or workflows, and then produces a file required by the application. Depending on the implementation of a data regeneration plug-in, a regenerated dataset may be copied at the end of the regeneration process, or, when streaming is enabled, the output data will be written to a buffer and the location of this buffer will be registered in a global file registry.

4.5. Dynamic data source selection

The framework implements a data selector, which is responsible for choosing the best data source dynamically at runtime. The data selector works closely with a resource monitor, which is responsible for getting resource conditions information by communicating to a resource monitoring plug-in.

The data selector and the resource monitor both work independently to the application process. The framework runs them in a separate thread (using pthread) for each Active Data file; a thread is created when the file is opened (ad_open) and is killed when the file is closed (ad_close). Locking mechanism is implemented to prevent multiple accesses to the files from different threads.

In each monitoring thread, a data selection loop is run; this loop runs continuously until the thread is killed. For every interval, the loop requests the information about the replica servers from the resource monitor. The data selector then chooses the best file source and notifies the system when a server change is needed. Then, the loop is in sleep mode until the next interval. Currently, a data source is selected by comparing the network latency, bandwidth and the processing load of the replica hosts.

5. Case study: an atmospheric science workflow

During the development of the ADS prototype, we have conducted a number of case studies [22][23][24][21]. In this section, an atmospheric workflow (Figure 13) is used to demonstrate the prototype’s unified data access interface; in particular, to evaluate the performance overheads of using a virtual file, and to test a Kepler data regeneration plug-in, which is capable of recreating one or more datasets by running a Kepler workflow at runtime.
This workflow consists of a number of software models. There is a global climate model, CCAM [31], which drives the boundaries of a regional weather model, DARLAM [32], which in turn produces data used by a photochemical pollution model, CIT [41]. In fact, these are standalone programs and each of them was developed for a particular purpose. Because the mesh solutions are different between the models, data needs to be refined. The cc2lam utility fine-grains the climate data for regional studies, while the lam2cit tool converts the netCDF [33] data produced by DARLAM to the IEEE format used by the CIT model.

Two Kepler workflows have been created; Figure 14 illustrates a workflow that runs each component sequentially using a SDF director and Figure 15 illustrates a workflow that runs in parallel with data stream pipes using a PN director. These workflows are stored and eventually used by the Kepler plug-in, which runs one of these workflows to create the data file (winds_progfc_6km.bin) used by the CIT model.
When the CIT model was executed during the case study, the interface discovered that an input file was missing. It queried the prototype and found that the file can be recreated by running a Kepler workflow. As part of the open operation, the interface invoked the data regenerator to recreate the data. After that, the regenerated file was used for subsequent file operations.

Figure 16 graphs the average execution time for both SDF and PN workflows. Since the PN director allows software components to run in parallel, the need of waiting for the completion of a driving model is eliminated; in this mode, a model can start computation as soon as sufficient data is available, such as a time step. Not surprisingly, the PN workflow runs significantly faster than the SDF counterpart, with over 25% increase in overall performance.

Nevertheless, the case shows that the data regeneration process resulted in a ~7% performance loss overall. We attribute this to two reasons. First, we found that Kepler loads a large number of libraries during startup and consumes large amount of memory (possibly because of the use of Java). Second, we believe that there is an overhead in querying and retrieving the metadata using the SRB API. In fact, the metadata is stored in a PostgreSQL database and access to the database is via an ODBC; overheads may have occurred in both interfaces.

This case study demonstrates the use of virtual files and shows that deleted data can be regenerated by our ADS prototype. In addition, the results show that there is a small overhead in the data regeneration process due to the long Kepler initialization and multiple metadata access layers.

6. Summary

In this chapter, we have described a Grid Data Life Cycle and the design of our Active Data model. We have also briefly discussed the implementation of a prototype and demonstrated how this prototype supports on-demand data regeneration using an atmospheric workflow.
References


The Evolution of Research and Education Networks and their Essential Role in Modern Science

William JOHNSTON1, Evangelos CHANIOTAKIS, Eli DART, Chin GUOK, Joe METZGER, Brian TIERNEY
ESnet, Lawrence Berkeley National Laboratory†

Abstract. ESnet – the Energy Sciences Network – has the mission of enabling the aspects of the US Department of Energy’s Office of Science programs and facilities that depend on large collaborations and large-scale data sharing to accomplish their science. The network requirements of this community have been explored in some detail by ESnet and a long-term plan has been developed in order to ensure adequate networking to support the science. In this paper we describe the planning process (which has been in place for several years and was the basis of a new network that is just now being completed and a new set of network services) and examine the effectiveness and adequacy of the planning process in the light of evolving science requirements.

Keywords. Energy Sciences Network (ESnet), networks for large-scale science, network planning.

Introduction

The US Department of Energy’s Office of Science (“SC”) is the single largest supporter of basic research in the physical sciences in the United States providing more than 40 percent of total funding for this area. See [1].

The Office of Science supports researchers at more than 300 colleges and universities across the United States. SC balances its support for big science and interdisciplinary teams with investments in basic research projects conducted by leading university and laboratory investigators. The DOE FY2006 university research funding exceeded SUS 800 million.

The National Laboratories are the heart of SC’s science programs (Advanced Scientific Computing Research, Basic Energy Sciences, Biological and Environmental Research, Fusion Energy Sciences, High Energy Physics, and Nuclear Physics). The DOE national laboratory system is the most comprehensive research system of its kind in the world – and the backbone of American science. The Office of Science is the steward of 10 of these 17 laboratories with world-class capabilities for solving complex interdisciplinary scientific problems. The 10 science Labs are Ames Laboratory, Argonne National Laboratory (ANL), Brookhaven National Laboratory (BNL), Fermi National Accelerator Laboratory (FNAL), Thomas Jefferson National Accelerator

† This work was supported by the Director, Office of Science, Office of Advanced Computing Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.
1 Corresponding Author: William Johnston, wej@es.net
Facility (JLab), Lawrence Berkeley National Laboratory (LBNL), Oak Ridge National Laboratory (ORNL), Pacific Northwest National Laboratory (PNNL), Princeton Plasma Physics Laboratory (PPPL) and SLAC National Accelerator Laboratory (SLAC).

At the Scientific User Facilities the Office of Science builds and operates some of the world’s most important scientific facilities and instruments that researchers depend on to extend the frontiers of science. The Office of Science facilities include particle accelerators, synchrotron light sources, neutron scattering facilities, nanoscale science research centers, supercomputers, high-speed networks, and genome sequencing facilities.

1. DOE Office of Science and ESnet – ESnet and its Mission

ESnet is an Office of Science facility in the Office of Advanced Scientific Computing Research (“ASCR”). ESnet’s primary mission is to enable the science goals of the Office of Science (SC) and that depend on:

- Sharing of massive amounts of data
- Thousands of collaborators world-wide
- Distributed data processing
- Distributed data management
- Distributed simulation, visualization, and computational steering

In order to accomplish its mission ESnet provides high-speed networking and various collaboration services to Office of Science laboratories as well as for many other DOE programs (on a cost recovery basis). The goal is to ensure that they have robust communication with their science collaborators in the U.S. and world-wide.

To this end ESnet builds and operates a high-speed national network specialized to serving tens of thousands of Department of Energy scientists, support staff, and collaborators worldwide. This network provides high-bandwidth, reliable connections from the DOE science community and to all of the world’s major research and education (R&E) networks that serve U.S. and international R&D institutions. This enables researchers at national laboratories, universities and other institutions to communicate with each other to accomplish the collaboration needed to address some of the world's most important scientific challenges.

The ESnet architecture and capacity are driven by DOE’s involvement in some of the world’s largest science experiments. The Large Hadron Collider (LHC) comes online in 2009, resulting in an urgent demand for guaranteed very high-bandwidth connectivity (greater than 10 Gigabits (10,000 Megabits/sec) for huge data transfers. Such service is also identified as required by other Office of Science mission areas as a result of a formal requirements gathering process. This process has shown that similar needs exist for the climate community in the near term, for ITER (international fusion reactor experiment) when it comes online a decade from now, for the huge volumes of data generated in the rapidly evolving SC scientific supercomputing centers and their next generation numerical models (e.g. for climate), and a new generation of instruments such as those associated with the study of dark matter/dark energy cosmology. As DOE’s large-scale science moves to a distributed national and international model, ESnet must provide the innovation and expertise to meet these networking needs.
To meet specific Office of Science requirements, in 2008 ESnet completed the initial rollout of ESnet4 (Figure 1) which involved deploying a 30 Gb/s core network on a footprint that covers the country in six interconnected rings – this network will grow to 50Gb/s by 2010. The overall ESnet network consists of two core networks that address two different types of traffic. The IP core network carries all of the “commodity” IP traffic from the Labs, including much of the small and medium scale science traffic. The IP core connects to commercial and R&E peering points. The SDN core is primarily designed to carry the large data flows of science and connects primarily to similar high-speed R&E networks (e.g. Internet2’s Dynamic Circuit Network and NLR’s FrameNet). However, under some circumstances the IP core will carry SDN traffic and the SDN core will carry “commodity” IP traffic. One 10 Gb/s wave/optical circuit is used for the IP core and all of the rest of the waves/optical circuits, including the ones added over the next several years, will be incorporated into the SDN core.

In addition to high-speed connectivity with all of the US research and education (R&E) networks, ESnet4 provides connections to high-speed networks in Europe, Japan and SE Asia, as well as to more than 100 other R&E networks around the world. This ensures that DOE researchers can have high-speed connectivity with all collaborating national and international researchers. In addition to the considerable end-to-end bandwidth and the high reliability of ESnet’s operational model, ESnet is actively involved with other domestic and international R&E networks in developing and deploying cutting-edge technologies and services needed to produce a seamless interoperable infrastructure that will allow the advancement of DOE’s large-scale science. ESnet services allow scientists to make effective use of unique DOE research facilities and computing resources, independent of time and geographic location.
SC science collaborators are those SC supported or SC facility using scientists who are not at the DOE Labs. This large community provides input to ESnet through numerous community meetings that ESnet participates in. This community is represented to ESnet primarily by the R&E network organizations (such as Internet2, DANTE/GÉANT, the U.S. regional networks, the European NRENS, etc.) that serve this science collaboration community.

2. Strategic Approach to Accomplishing ESnet’s Mission

ESnet has a three-pronged approach to planning for meeting its mission:

I. Determine the ESnet requirements by a) working with the SC community to identify the networking implication of the instruments, supercomputers, and the evolving process of how science is done, and b) considering the historical trends in the use of the network.

II. Develop an approach to building a network environment that will enable the distributed aspects of SC science and then continuously reassess and update the approach as new requirements become clear.

III. Anticipate future network capabilities that will meet future science requirements with an active program of RESEARCH and advanced development.

2.1. Strategy I: Requirements from examining SC plans for future instruments and science and from historical trends in the network

Science requirements are determined by considering two aspects of the origins of network traffic. First, by exploring the plans and processes of the major stakeholders – the data characteristics of scientific instruments and facilities (what data will be generated by instruments and supercomputers coming on-line over the next 5-10 years?) and by examining the future process of science (how and where will the new data be analyzed and used, what sorts of distributed systems are involved, etc. – that is, how will the process of doing science change over 5-10 years?). And second, by observing current and historical network traffic patterns – what do the trends in network patterns predict for future network needs?

2.1.1. Examine SC plans for future instruments and science

The primary mechanism for determining science community plans is through the six Office of Science Network Requirements Workshops. These workshops are organized by the SC Program Offices and are conducted at the rate of two requirements workshops per year, repeating starting in 2010.

Additionally several earlier workshops (2002/3 and 2006) looked at specific HEP facilities and ASCR supercomputer centers†. The Workshops have now examined a fair cross section of the Country’s major science facilities operated by DOE’s Office of Science.

† The workshop reports are available at www.es.net/hypertext/requirements.html
2.1.2. Requirements from the Workshops

Table 1 summarizes the quantitative and functional requirements that have been determined from the Workshops to date. The information represents a mix of the characteristics of the instruments / systems involved and the process of the associated science – that is, how the instrument / system is used, by whom, the size of the collaborations, the locations of the collaborators, etc. The reflected bandwidths are generally aggregate bandwidths, so they lack the necessary level of detail to actually build out the network infrastructure to get to specific locations. This issue is addressed below.

Table 1. Quantitative and functional requirements that have been determined from the requirements workshops to date.

<table>
<thead>
<tr>
<th>Science Drivers</th>
<th>End2End Reliability</th>
<th>Near Term End2End Band width</th>
<th>5 years End2End Band width</th>
<th>Traffic Characteristics</th>
<th>Network Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCR: ALCF supercomputer</td>
<td>-</td>
<td>10Gb/s</td>
<td>30Gb/s</td>
<td>• Bulk data</td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote control</td>
<td>• Deadline scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote file system sharing</td>
<td>• PKI / Grid</td>
</tr>
<tr>
<td>ASCR: NERSC supercomputer</td>
<td>-</td>
<td>10Gb/s</td>
<td>20 to 40Gb/s</td>
<td>• Bulk data</td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote control</td>
<td>• Deadline scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote file system sharing</td>
<td>• PKI / Grid</td>
</tr>
<tr>
<td>ASCR: NLCF supercomputer</td>
<td>-</td>
<td>Backbone Bandwidth Parity</td>
<td>Backbone Bandwidth Parity</td>
<td>• Bulk data</td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote control</td>
<td>• Deadline scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote file system sharing</td>
<td>• PKI / Grid</td>
</tr>
<tr>
<td>BER: Climate</td>
<td>-</td>
<td>3Gb/s</td>
<td>10 to 20Gb/s</td>
<td>• Bulk data</td>
<td>• Collaborative services</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Rapid movement of GB sized files</td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote Visualization</td>
<td>• PKI / Grid</td>
</tr>
<tr>
<td>BER: EMSL/Bio</td>
<td>-</td>
<td>10Gb/s</td>
<td>50-100Gb/s</td>
<td>• Bulk data</td>
<td>• Collaborative services</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Real-time video</td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote control</td>
<td>• PKI / Grid</td>
</tr>
<tr>
<td>BER: JGI/Genomics</td>
<td>-</td>
<td>1Gb/s</td>
<td>2-5Gb/s</td>
<td>• Bulk data</td>
<td>• Dedicated virtual circuits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Guaranteed bandwidth</td>
</tr>
<tr>
<td>BES: Chemistry and Combustion</td>
<td>-</td>
<td>5-10Gb/s</td>
<td>30Gb/s</td>
<td>• Bulk data</td>
<td>• Data movement middleware</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Real time data streaming</td>
<td></td>
</tr>
<tr>
<td>BES: Light Sources</td>
<td>-</td>
<td>15Gb/s</td>
<td>40-60Gb/s</td>
<td>• Bulk data</td>
<td>• Collaboration services</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Coupled simulation and experiment</td>
<td>• Data transfer facilities</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Grid / PKI</td>
</tr>
<tr>
<td>BES: Nanoscience Centers</td>
<td>-</td>
<td>3-5Gb/s</td>
<td>30Gb/s</td>
<td>• Bulk data</td>
<td>• Collaboration services</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Real time data streaming</td>
<td>• Grid / PKI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Remote control</td>
<td></td>
</tr>
</tbody>
</table>
The last four entries in the list for High Energy Physics and Nuclear Physics, HEP in particular, provided many of the specific requirements in 2004-5-6 that were the basis of the detailed design of ESnet4.

Requirements for services emerge from detailed discussions about how the analysis and simulation systems actually work: where the data originates, how many systems are involved in the analysis, how much data flows among these systems, how complex is the work flow, what are the time sensitivities, etc.

The result of this process is that fairly consistent requirements are found across the large-scale sciences. Large-scale science uses distributed systems in order to:

- couple existing pockets of code, data, and expertise into “systems of systems”
- break up the task of massive data analysis into elements that are physically located where the data, compute, and storage resources are located
Such systems

- are data intensive and high-performance, typically moving terabytes a day for months at a time
- are high duty-cycle, operating most of the day for months at a time in order to meet the requirements for data movement
- are widely distributed – typically spread over continental or inter-continental distances
- depend on network performance and availability, but these characteristics cannot be taken for granted, even in well run networks, when the multi-domain network path is considered

The nodes of the distributed systems must be able to get guarantees from the network that there is adequate bandwidth necessary to accomplish the task at hand. The systems must also be able to get information from the network to support graceful failure and auto-recovery and adaptation due to unexpected network conditions that are short of outright failure.

**General Network Technology and Capabilities Requirements from Instruments and Facilities**

Considering the overall requirements from the Workshops one can identify a generic, but important, set of goals for any network and network services implementation:

- **Bandwidth**: Adequate network capacity to ensure timely movement of data produced by the facilities
- **High reliability** is required for large instruments and “systems of system” (large distributed systems) which now depend on the network to accomplish their science
- **Connectivity**: The network must have the geographic reach – either directly or through peering arrangements with other networks – sufficient to connect users and collaborators and analysis systems to SC facilities
- **Services** that provide guaranteed bandwidth, traffic isolation, end-to-end monitoring, etc., are required and these services must be presented to the users in a framework of Web Services / SOA (service oriented architecture) / Grid / “Systems of Systems” that are the programming paradigms of modern science.

**Connectivity**

Much of the design of the ESnet network footprint is intended to accommodate the high degree of connectivity that is needed to meet the requirements of the science collaborations. The footprint must not only accommodate connecting the approximately 50 ESnet sites, but also ensure that high-speed connections can be made to all of the major U.S. and international R&E networks. Although much of this is accomplished by ensuring a presence at the half dozen or so R&E exchange points (MAN LAN in New York, MAX in Washington, DC, Starlight in Chicago, PNWGigaPoP in Seattle, etc.) even at those locations there may be several local points of presence that must be accommodated. The major international locations associated with SC programs are illustrated in the next figure.
Requirements for connectivity: The international collaborators of DOE’s Office of Science, drives ESnet design for international connectivity.

Other Requirements

Assistance and services are needed for smaller user communities that have significant difficulties using the network for bulk data transfer. Part of the problem here is that WAN network environments (such as the combined US and European R&E networks) are large, complex systems like supercomputers, and it is simply unrealistic expect to get high performance when using this “system” in a “trivial” way – this is especially true for transferring significant amounts of data over distances of thousands to tens of thousands of km.

Some user groups need more help than others.

• Collaborations with a small number of scientists typically do not have network tuning expertise
  – They rely on their local system and network administrators
  – They often don’t have much data to move
  – Their data transfer and storage systems typically have default network tuning parameters and typically suffer poor wide area data transfer performance as a result
  – Therefore, they avoid using the network for data transfer if possible

• Mid-sized collaborations have a lot more data, but similar expertise limitations
  – More scientists per collaboration, much larger data sets (10s to 100s of terabytes)
  – Most mid-sized collaborations still rely on local system and networking staff, or supercomputer center system and networking staff for WAN assistance (where their expertise is typically limited)

• Large collaborations (HEP, NP) are big enough to have their own internal software shops
  – Dedicated people for networking, performance tuning, etc
  – Typically need much less assistance in using wide area networks effectively
Often held up (erroneously) as an example to smaller collaborations

A semi-quantitative assessment of this situation based on experience is illustrated in Figure 3.

Figure 3. Rough user grouping by collaboration data set size. (Eli Dart, ESnet, Don Petravick, FNAL/DOE-SC-HEP).

To address some of these issues ESnet has established a Web site for information and best practice on system tuning for high wide area network throughput: fasterdata.es.net.

Observations from the Requirements Workshops

The concept that that grew out of early work with the HEP community – namely that modern science cannot be accomplished without capable, high quality, very high-speed networks interconnecting the collaborators, their instruments, and their data repositories – has been validated, extended, and nuanced based on the findings of the first four (of six) formal requirements workshops.

These workshops use case studies to identify how instruments (including supercomputers) and the process of doing science are changing and how those changes both depend on and drive the network.

One of the fundamental strategic realizations from HEP, and confirmed by the workshops, is that science productivity is now directly related to the amount of data that can be shared / accessed / incorporated / processed / catalogued in the new process of science that is heavily dependent on data analysis and computational models used by individuals, groups, and large collaborations of scientists. This is directly evident from the workshops in the areas of combustion and climate simulation, protein and cellular modeling (Proteomics), and the collaborative data analysis associated with large scientific instruments such as STAR and PHENIX at RHIC and Atlas and CMS at the LHC.
Several observations come from these sorts of specific science use of networks case studies:

- Networks are critical infrastructure for science, but no scientist wants to have to be aware of the network.
- Networks are essential for the functioning of large experiments, but the cost of networks (and computing) is tiny compared to the cost of large instruments (networks typically cost millions of dollars where scientific instruments often cost many billions of dollars – the LHC is a roughly $US 10 billion experiment)
- Networks are just one part of the infrastructure of data management that is key for scientific productivity – middleware that provides semantically relevant operations for data management is critical to provide a more abstract and capable data movement functionality
- For science data movement, all that matters is end-to-end (application to application) performance. The network could be “infinitely” fast and if some other problem (e.g. access to end host or storage) limits throughput to 100 Kb/s, the fast network is almost useless.
- Middleware is critical, but no matter how good it is it will not be effective if it does not fit into the “culture” of any given science community – utility and usability are critical.
- Different science communities have different requirements and different approaches to data movement / sharing. There will be (are) many different types of data movement middleware. This results in a problem that is too diffuse / too big for a “user” assistance team in a networking organization to have much impact on the overall scientific community. (Network organizations like ESnet and Internet2 are very small compared to many scientific collaborations – especially those involving large instruments.) Therefore it may be that all that the networking organizations can do is to try and provide general advice: “best practice” prescriptions for what approaches work, cautions about what not to do, goals for what should be achievable, etc. In other words collect and publicize best practices at several levels. One example of this is ESnet’s WAN performance tuning site fasterdata.es.net.
- Just the process of doing outreach, as in ESnet’s requirements workshops, can identify potential or extant issues which leads to increased user awareness, which generates conversations about possible mitigation strategies. This frequently results in the science community beginning a process of integrating WAN data movement throughput solutions into the science planning.

2.1.3. Science requirements determination: Requirements from observing current and historical network traffic patterns

ESnet has detailed historical network traffic information that is accurate going back to about 1989. There is a lot of history of the evolution of science use of networking summarized in this data.
By looking at some of the implications of the historical traffic (Figure 4), together with some link-specific traffic data, we see that in 4 years we can expect a 10x increase in traffic over current levels just based on historical trends:

- nominal average load on the busiest backbone links in June 2006 was ~1.5 Gb/s
- in 2010 average load will be ~15 Gb/s based on current trends and 150 Gb/s in 2014

(These projections are based on early 2006 data were made just as the large-scale science begin to dominate ESnet traffic and so are certainly understated.)

Measurements of this type are science-agnostic – it doesn’t matter who the users are or what they are doing: the traffic load is increasing exponentially. Predictions based on this sort of forward projection could produce low estimates of future requirements because they cannot entirely predict new uses of the network (though some of this is built into the “memory” of past uses of the network, which includes new instruments coming on line).

**Large-Scale Science Now Dominates ESnet**

ESnet carries about 2 x 10⁹ flows (connections) per month which account for all of the ESnet traffic. Large-scale science – LHC, RHIC, climate data, etc. – now account for about 90% of all ESnet traffic, but in only a few thousand flows/month. What this implies is that a few large data sources/sinks now dominate ESnet traffic, which means that in the future, overall network usage will follow the patterns of the very large users. (Figure 5) Managing this situation in order to provide reliable and predictable service to large users while not disrupting a lot of small users, requires the ability to isolate these flows to a part of the network designed for them (“traffic engineering”).
2.2. Strategy II: Develop an approach to building a network environment that will enable the distributed aspects of SC science and then continuously reassess and update the approach as new requirements become clear

ESnet has evolved a planning process that involves:

1. Providing the basic, long-term bandwidth requirements with an adequate and scalable infrastructure;
2. Undertaking continuous reexamination of the long-term requirements because they frequently change;
3. Identifying required new network services and implement them;
4. Developing outreach approaches that identify what communities are not being well served by the network because they are not equipped to make effective use of the network.

To elaborate:

1. Providing the basic, long-term bandwidth requirements with an adequate and scalable infrastructure:

The HEP requirements as articulated in 2004/5 were the basis of the architecture, capacity (and therefore budget), and services of the next generation network (ESnet4). However, because the aggregate HEP requirements were large and the specific connectivity requirements were geographically diverse, ESnet was able to design a network that was generically “large,” relatively high aggregate capacity, had a comprehensive network footprint, and was flexible and scalable in several dimensions. The implementation of the first phase of ESnet4 (based on an Internet2-ESnet
partnership for the underlying optical network) was done from mid-2006 to late 2008 (now complete – see Figure 1). This approach of “a rising tide lifts all boats” has proven to be effective in addressing most of the network requirements of the other SC programs that have been identified after the 2004/5 planning process.

2. Undertaking continuous reexamination of the long-term requirements because they frequently change:

In spite of a methodical approach to determining the long-term requirements (mostly via the requirements workshops), unanticipated requirements show up in what appears to be an inherently hap-hazard way, with surprises coming from all quarters. There are two probable reasons for this:

- The SC science Programs are large and while the workshops try and characterize the major science in each program some areas are missed
- The processes of science are constantly refined and updated which frequently produces new requirements

Is there sufficient flexibility and scalability in the original “new network” (ESnet4) design and implementation to accommodate the “surprises”? Probably, but not certainly.

3. Identify required new network services and implement them:

Intuiting the actual, useful implemented form of a new network service from the user articulated requirements (e.g. guaranteed bandwidth) is very hard. It runs the danger of using detailed requirements from too small a portion of the community (probably necessary initially in order to implement anything) and subsequently discovering that one community’s service semantic does not solve for another community what appeared to be the same issue.

Further, no network that is useful to the science community operates in isolation. Almost any new service that is going to be useful must be implemented in all of the networks that are involved in (typically) global science collaborations. This means that most of the world’s R&E networks must agree on the service description and then implement it in the particular architecture and implementation (hardware + circuits) of their network. Just getting the required level of coordination – even after there is in-principle agreement that the service is necessary – is a labor intensive and lengthy process. Even so, in two important areas – the guaranteed bandwidth / circuit services and end-to-end monitoring – this cooperative process is working well.

4. Develop outreach approaches that identify what communities are not being well served by the network because they are not equipped to make effective use of the network:

Even when the network tries to deal with particular communities (science collaborations) the number of different solutions (typically middleware that not only solves the network interface problem but works with all of the applications that the community already has developed) is of a scale that rapidly exceeds the manpower that can reasonably be made available in the network organizations. This is also a lengthy process that has to be approached with considerable “social skills” in order to be effective in raising the community’s awareness to the point that they see that science resources must be allocated to the problem of effective use of the network.
2.2.1. Planning Process 1. Provide the basic, long-term bandwidth requirements with an adequate and scalable infrastructure

ESnet4 was built to address specific Office of Science program requirements. The result is a much more complex and much higher capacity network than in the past.

Building the Network as Opposed to Planning the Budget

Aggregate capacity requirements like those shown in Table 1 indicate how to budget for a network but do not tell you how to build a network. To actually build a network you have to look at where the traffic originates and ends up and how much traffic is expected on specific paths. In 2006, when the construction of ESnet4 started, we had more or less specific bandwidth and path (collaborator location) information for LHC (CMS, CMS Heavy Ion, and Atlas), the SC Supercomputers, CEBF/JLab, and RHIC/BNL. This specific information has lead to the current and planned configuration of the network for the next several years. (Figure 6 illustrates the resulting path configuration and the anticipated path loadings (Gb/s) in 2010.)

Figure 6. The SC facilities core network path and bandwidth requirements for 2010 as identified in 2006. The network in 2008 has two 10Gb/s optical circuits on the footprint illustrated. One 10G circuit will be added each year until there are a total of six – the current planned capacity of ESnet4.

As a gauge of how realistic the 40-50 Gb/s traffic predictions are we can look at the full-scale analysis system trials conducted by the LHC, CMS detector collaboration. During one four month period these trials were generating 9 Gb/s of traffic – about 4 Gb/s average sustained over the entire four months (Figure 7). This clearly indicates that 1) the bandwidth predictions are not unreasonably high, and 2) that a new generation of scientific instruments and distributed data analysis is generating levels of network traffic never before seen in the R&E community.
One (deliberate) consequence of ESnet’s new architecture is that site availability (as seen from other sites) is increasing. (0)
2.2.2. Effectiveness of the approach: Re-evaluating the Strategy and Identifying Issues

The current strategy (that lead to the ESnet4, 2010-12 plans) was developed primarily as a result of the information gathered in the 2002 and 2003 network workshops, and their updates in 2005-6 that had input from the LHC, climate, RHIC, SNS, Fusion Energy sites, supercomputing, and a few others) [2]. So far, the more formal requirements workshops have largely reaffirmed the ESnet4 strategy developed based on these early workshops. However – is this the whole story?

How do the Science Program Identified Requirements Compare to this Capacity Planning?

We can get a feeling for whether the planning process is adequate, at least in aggregate, by looking at the aggregate requirements as articulated and comparing them to 1) the planned network, 2) an estimate of the growth rates of the planned capacity and the projected growth of science data.

Table 2 shows the aggregate bandwidth requirements (“5 year end-to-end”) from the general descriptions of the instruments and the amount of data that the science community knows will have to be put onto the network (from Table 1); the bandwidth accounted for in path planning (because there was enough information available about who would be using the data to define network paths for that data), and; the difference (requirements unaccounted for in the actual network path planning).

<table>
<thead>
<tr>
<th>Science Areas / Facilities</th>
<th>5 year end-to-end bandwidth requirements (Gb/s)</th>
<th>accounted for in current ESnet path planning</th>
<th>Unacct’ed for</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCR: ALCF</td>
<td>30</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>ASCR: NERSC</td>
<td>40</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>ASCR: NCLF</td>
<td>50</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>BER: Climate</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>BER: EMSL/Bio</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>BER: JGI/Genomics</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>BES: Chemistry and Combustion</td>
<td>30</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>BES: Light Sources</td>
<td>60</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>BES: Nanoscience Centers</td>
<td>30</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Fusion: International Collaborations</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Fusion: Instruments and Facilities</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Fusion: Simulation</td>
<td>88</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>HEP: LHC</td>
<td>265</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>NP: CMS Heavy Ion</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>NP: JLAB</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>NP: RHIC</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>789</strong></td>
<td><strong>405</strong></td>
<td><strong>384</strong></td>
</tr>
</tbody>
</table>
This difference was anticipated in the network design and the planned (and budgeted for) network capacity is considerably larger than the 405 Gb/s “accounted for” plans, and appears adequate to meet the projected needs (Table 3).

Table 3. ESnet Planned Aggregate Capacity (Gb/s) Based on 5 yr. Budget vs. 5 yr. Science Network Requirements Aggregation Summary.

<table>
<thead>
<tr>
<th>year</th>
<th>2006</th>
<th>2007</th>
<th>2008</th>
<th>2009</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESnet planned aggregate capacity</td>
<td>57.50</td>
<td>192</td>
<td>192</td>
<td>842</td>
<td>1442</td>
<td>1442</td>
<td>1442</td>
<td>2042</td>
</tr>
<tr>
<td>Requirements (aggregate Gb/s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>789</td>
</tr>
</tbody>
</table>

Note that the “planned aggregate capacity” measure is the sum of the link capacity on all of the major inter-city links in the core network. There is enough peering point and topological diversity (that is, there is traffic flowing into and out of the network in almost every one of the cities that define the inter-city links) that this has proven to be at least a somewhat useful measure.

Therefore, the planned aggregate capacity growth of ESnet matches the know requirements as understood in 2006/8. The “extra” capacity indicated in the table (e.g. 2042 Gb/s planned vs. 789 Gb/s required in 2013) is needed to account for the fact that there is less than complete flexibility in mapping specific path requirements to the aggregate capacity planned network and specific paths specific paths will not be known until several years into building the network. Whether this approach works is to be determined, but indications are that it probably will.

2.2.3. Planning Process 2. Undertaking continuous reexamination of the long-term requirements because they frequently change

Recently ESnet has undertaken an effort to identify trends that would, at least somewhat independently, validate or change the projections developed by the methodologies described above. This was done by a combination of soliciting user estimates and looking at the 15 year historical traffic trends in the network.

One hypothesized way to develop an independent measure of growth is to assume that the science processes of large instruments (and probably supercomputers in the next few years) have changed forever and large scale distributed collaborations are now the norm. Because of this, just looking at the growth of the volume of data that must be managed by any discipline will provide a measure of the growth of network traffic.

As an indication of the validity of this assumption (the changing process of science is continuing to drive more use of the network) consider what is happening to high energy physics data analysis.

When the network planning was done for the two major LHC experiments (CMS and ATLAS) in 2005-6 the assumption (by the HEP community) was a fairly strict hierarchical model with the LHC at the apex (tier 0) sending data to nation-based tier 1 data centers (e.g. FNAL and BNL in the U.S.). The data analysis would mostly be done at tier 2 centers (universities with significant computing capability) and that the tier 2 centers would mostly generate 3-5 Gb/s network flows from one tier 1 center. Tier 3 centers (smaller universities) would get data mostly form the tier 2 centers at about 1 Gb/s.

However, what has happened is that several Tier2 centers are capable of 10Gb/s now and many Tier2 sites are building their local infrastructure to handle 10Gb/s.
Further, many Tier3 sites are also building 10Gb/s-capable analysis infrastructures. Both Tier2 and Tier3 sites will be accessing data from multiple Tier1 data centers. This was not in LHC plans even a year ago.

It has been said§ that scientific productivity will follow high-bandwidth access to large data volumes and this provides a powerful incentive for science groups to upgrade network capacity and to use it.

We are already seeing the “onslaught” of this traffic from the Tier 2 centers (not anticipated in 2006) and it now seems likely that this will cause a second onslaught in 2009 as the Tier3 sites all upgrade their network and computing capacity to handle 10Gb/s of LHC traffic. In other words it is possible that the USA installed base of LHC analysis hardware will consume significantly more network bandwidth than was originally estimated.

There are already indications of this when looking at the historical traffic record in a little more detail than just the 15 year aggregate. If we project that traffic from 1990 to 2004 (which we refer to as the “old” era vs. “new” era because of the clear change in traffic patterns in 2004-5 (0)) as one projection and the traffic from 2004-2008 as a second projection, what we see is that, indeed, there is a significant difference, especially as the projections are on a log plot. (Figure 9)

As another indicator, if we consider the growth of the volume of scientific data that will almost certainly be moved over the network in the future (as it is today in the high energy physics community) we had a basis for comparing this growth with the planned capacity of the network. The result (Figure 10) is that three measures (projected traffic, projected HEP data, and projected climate simulation data) are growing faster than the planned capacity of the network.

§ Harvey Newman (HEP, Caltech) predicted this eventuality several years ago.

![Figure 9. Projected Aggregate Network Utilization: New Era vs. Old.](image-url)
Figure 10. Projected science data volume, ESnet traffic, and ESnet capacity. (Ignore the units of the quantities being graphed they are normalized to 1 in 1990, just look at the long-term trends: All of the “ground truth” measures are growing significantly faster than ESnet projected capacity.) (HEP data courtesy of Harvey Newman, Caltech, and Richard Mount, SLAC. Climate data courtesy Dean Williams, LLNL, and the Earth Systems Grid Development Team.).

Issues for the Future Network

The current estimates from the LHC experiments and the supercomputer centers have the currently planned ESnet 2011 wave configuration operating at capacity and there are several other major sources that will be generating significant data in that time frame (e.g. Climate). The significantly higher exponential growth of traffic (total accepted bytes) vs. total capacity (aggregate core bandwidth) means traffic will eventually overwhelm the capacity – “when” cannot be directly deduced from aggregate observations, but if you add the fact that the nominal average load on busiest backbone paths in June 2006 was ~1.5 Gb/s - In 2010 average load will be ~15 Gb/s based on current trends and 150 Gb/s in 2014 then one could guess that capacity problems will start to occur by 2015-16 without increasing the planned capacity of the network. Further, the “casual” increases in overall network capacity based on straightforward commercial channel capacity that have sufficed in the past are less likely to easily meet future needs due to the (potential) un-affordability of the hardware. For example the few existing commercial examples of >10G/s router/switch interfaces are ~10x more expensive than the 10G interfaces which is not a practical solution unless prices come down.

Further, we cannot just keep adding capacity by provisioning more 10 Gb/s optical circuits for several reasons. First is that the cost of the sort of managed wave (circuit) service that ESnet uses (and feels is necessary for the required level of reliability based on observing the reliability of wide area networks are operated by the R&E community) is not practical for two reasons. A 10 Gb/s circuit across ESnet’s entire footprint (as in the planned network) is more than $US 1 million/year. Second, there are not enough waves available to the R&E community. The several national fiber
footprints that exist today are unlikely to be able to increase because of projected capacity demands by the telecom industry.

*Where Will the Capacity Increases Come From?*

This leaves us with two likely approaches to increase capacity.

ESnet4 planning assumes technology advances will provide 100 Gb/s optical waves (they are 10 Gb/s now) which gives a potential 5000 Gb/s aggregate (in the sense of summing link capacities as described above) core network by 2012. The ESnet4 SDN switching/routing platform is designed to be upgradable to 100 Gb/s network interfaces, so not all of the core network equipment would have to be replaced at the same time. With capacity planning based on the ESnet 2010 wave count, together with some considerable reservations about the affordability of 100 Gb/s network interfaces, we can probably assume some fraction of the 5000 Gb/s of potential, aggregate core network capacity by 2012 depending on the cost of the equipment – perhaps 20% – about 2000 Gb/s of aggregate path capacity (which is the assumption in the number in 0 and 0.5.2293857.1311136). Increases beyond this will depend on 100G waves, and the associated network equipment, becoming affordable.

The second approach involves using a more dynamic use of the underlying optical infrastructure than is possible today.

The Internet2-ESnet partnership optical network that both organization use today is build on dedicated fiber and optical equipment that is configured with 10 X 10G waves / fiber path. The optical equipment allows for more waves to be added in groups of 10 up to 80 waves.

The current wave transport topology is essentially static or only manually configured – our current network infrastructure of routers and switches assumes this. However, assuming that some fraction of the 80 waves can be provisioned (i.e. at an affordable cost), then with completely flexible traffic management extending down to the optical transport level we should be able to extend the life of the current infrastructure by moving significant parts of the capacity to the specific routes where it is needed. This entails integrating the management of the optical transport with the “network” and providing for dynamism / route flexibility at the optical level in order to make optimum use of the available capacity. To understand the issues and the approach consider the next several figures.

Figure 11 illustrates the configuration of the optical node in the Internet2-ESnet network today. A fixed set of waves is assigned to each organization which then uses those waves as static circuits between interfaces on routers and switches.

Figure 12 illustrates the fact that, even though the Infinera optical nodes have internal wave switching capability, currently the path of a wave through the network is static – the same as though it were a physical circuit.

** Unlike a year ago, there are now some encouraging indicators regarding the possibility of affordable 100G circuits. For example, at the Supercomputing 2008 conference Internet2, ESnet, Infinera, Ixia, Juniper Networks and Level 3 Communications will aggressively develop and test emerging 100 Gigabit Ethernet (GbE) technologies over the next year with the aim of deploying them on the nationwide Internet2 and ESnet networks. Further, ESnet is now (7/2009) funded to build a 100 Gb/s testbed across the country.
Dynamically Routed Optical Circuits for Traffic Engineering

The Infinera optical devices ("DTN") convert all network traffic (Ethernet or SONET) to G.709 framing internally and the DTNs include a G.709 crossbar switch that can map any input (user network facing) interface to any underlying wave and therefore to any other network interface. By adding a layer 1 (optical) control plane that is managed
by Internet2 and ESnet, the G.709 switch can be controlled in concert with topology information communicated to the switches and routers, and the underlying topology of the optical network can be changed as needed for traffic management, including the use of topological path redundancy for increasing point to point capacity (Figure 13).

This layer 1 control plane is in the planning phase and a testbed to develop the control plane management that is isolated from the production network is in the late stages of planning. The control plane manager will be based on an extended version of the OSCARS [3] dynamic circuit manager.

Figure 13. Dynamically routed optical circuits allow for traffic engineering (load balancing / rerouting).

As more waves are provisioned in the Internet2-ESnet optical network these new waves can be shared and dynamically allocated to maximize the utilization of the available waves. This has the potential to double or triple the effective capacity of the network because of the rich and redundant physical topology.

Conclusions

The warning of the trends illustrated in Figure 10 (“Projected science data volume, ESnet traffic, and ESnet capacity.”) is that even though you cannot make direct predictions as to the traffic resulting from the science data volume increases; it is growing exponentially faster than the projected network capacity which is guaranteed to cause problems in the future. (My guess, based on looking at traffic growth and link loadings, is that problems will occur by 2015, or so, if nothing is done – WEJ).

On the other hand, there are two technology advances, both of which are being pursued – 100G optical circuits and dynamic wave / optical circuit management – that should provide new network capabilities that will carry us to the next full new technology generation. (And we have no particular insights into what that will look like, but it will be based on research going on in physics laboratories today, probably related to new uses of quantum mechanical effects.)
2.2.4. Planning Process 3. Identify required new network services and implement them

The capabilities that are users’ top priority are reservable, guaranteed bandwidth and user-level, end-to-end monitoring. The monitoring capability is being done as an international community effort to refine and deploy perfSONAR and is described in [4] and [5]. The reservable, guaranteed bandwidth service is described here.

OSCARS: ESnet’s Guaranteed Bandwidth Virtual Circuit Services

To support large-scale science, networks must provide communication capability that is service-oriented. In the case of the virtual circuit service, the capabilities must be:

- Configurable – Be able to provide multiple, specific “paths” (specified by the user as end points) with specific characteristics.
- Schedulable – Premium service such as guaranteed bandwidth will be a scarce resource that is not always freely available, therefore time slots obtained through a resource allocation process must be schedulable.
- Predictable – A committed time slot should be provided by a network service that is not brittle, i.e. reroute in the face of network failures is important.
- Reliable – The service should be transparently self-correcting, e.g. virtual circuit reroutes should be largely transparent to the user.
- Informative – When users go to configure distributed systems they (or their cyber agents) should be able to see average path characteristics, including capacity, etc. When things do go wrong, the network should report back to the user in ways that are meaningful to the user so that informed decisions can about alternative approaches.
- Scalable – The underlying network should be able to manage its resources to provide the appearance of scalability to the user.
- Geographically comprehensive – The R&E network community must act in a coordinated fashion to provide this environment end-to-end.
- Secure – The use of guaranteed virtual circuits consumes a valuable resource (guaranteed bandwidth) and connects directly into the user environment. Therefore the user must have confidence that the other end of the circuit is connected where it is supposed to be connected and that the circuit cannot be “hijacked” by a third party while in use.
- Provide traffic isolation – Users want to be able to use non-standard/aggressive protocols when transferring large amounts of data over long distances in order to achieve high performance and maximum utilization of the available bandwidth.

The ESnet Approach for Required Virtual Circuit Capabilities

ESnet’s OSCARS system provides configurability, schedulability, predictability, and reliability with a flexible virtual circuit (VC) service: user specifies end points, bandwidth, and schedule and OSCARS can dynamically traffic engineer the underlying paths. See [3].

Providing useful, comprehensive, and meaningful information on the state of the paths, or potential paths, to the user is based on perfSONAR, and associated tools. PerfSONAR provides real time information in a form that is useful to the user (via appropriate abstractions) and that is delivered through standard interfaces that can be
incorporated into SOA/Web Services style applications. Techniques still need to be
developed to monitor virtual circuits based on the approaches of the various R&E
nets – e.g. MPLS in ESnet, VLANs, TDM/grooming devices (e.g. Ciena Core Directors),
etc., and then integrate this into a perfSONAR framework.

Reliability approaches for Virtual Circuits are still under investigation and are
topics for R&D – there are many ramifications to “blindly” rerouting a VC and the
subtleties (e.g. how to minimally interfere with other circuits) need to be worked out.

Scalability will be provided by new network services that, e.g., provide dynamic
wave allocation at the optical layer of the network in order to increase the effective
bandwidth between end points by taking advantage of the path redundancy possible in
a topologically rich core network (see section “Where Will the Capacity Increases
Come From?,” above), and reduce routing overhead.

Geographic ubiquity of the services can only be accomplished through active
collaborations in the global R&E network community so that all sites of interest to the
science community can provide compatible services for forming end-to-end virtual
circuits. To accomplish this active and productive collaborations exist among numerous
R&E networks: ESnet, Internet2, Caltech, DANTE/GÉANT, some European NRENs,
some U.S. regional networks (RONs), etc.

OSCARS circuits are “secure” to the edges of the network (the site
boundary/DMZ) because they are managed by the control plane of the network which
is highly secure and isolated from the general traffic.

The end-to-end provisioning of OSCARS virtual circuits is provided by explicit
Label Switched Paths (LSP) using MPLS (Multi-Protocol Label Switching) and RSVP
(Resource Reservation Protocol). The explicit LSPs that are automatically traffic
engineered to prefer paths on the SDN supports both layer 3 (IP) and layer 2 (Ethernet
VLANs) end-user VC services. From a user’s perspective, the layer 3 VC service
provides an isolated IP “tunnel”, and the layer 2 VC service provides an Ethernet
VLAN path, both of which have bandwidth guarantees. Having the ability to traffic
engineer LSPs over both the SDN and IP has the advantage that virtual circuits can be
set up to sites that only have IP connections to the network, which is true for many
smaller institutions.

For guaranteed bandwidth there are several ways that the requirements may be met.
Most R&E networks use a hardware device (sometimes called a “grooming device”) that
provides the required capabilities between hardware interfaces on an Ethernet or
SONET based path. ESnet took a software approach based on several standard
protocols in order to provide end-to-end virtual circuits. OSCARS uses MPLS and
RSVP with QoS to provide guaranteed bandwidth within the SDN and IP core network.
MPLS and RSVP is used in provisioning explicit LSPs which are traffic engineered
hop-by-hop between the ESnet ingress and egress edge points. QoS is used to ensure
service guarantees (or lack thereof) for different traffic classes, such as guaranteed VC,
and best effort IP traffic. Policing on a per VC basis is at the ESnet ingress edge points
to prevent over-subscription of reservations, by ensuring that each user can only get the
requested and reserved bandwidth. Reservations are tracked link by link over time by
OSCARS to prevent over-booking of available link bandwidth (admission control).

\*\* Traffic grooming is the process of grouping many small network flows into larger units, which
can be processed as single entities. Dedicated hardware circuits such as Ethernet circuits are used
to connect the grooming devices together.
Configurability, schedulability, predictability (bandwidth guarantees), and reliability needs identified in the science requirements are part of OSCARS by design since this is the problem that OSCARS was intended to address.

**OSCARS Status**

ESnet developed the original OSCARS code base working with Internet2. In the ESnet centric deployment of the service in the network, a prototype layer 3 (IP) guaranteed bandwidth virtual circuit service was deployed in ESnet in early 2005. A prototype layer 2 (Ethernet VLAN) virtual circuit service deployed in ESnet in third quarter 2007. Support for “soft reservations” (reservations that are guaranteed the user requested bandwidth under congestion scenarios but can burst higher if excess bandwidth) was provided in the second quarter 2008. Automatic diagramming of VCs associated with a given site was introduced in second quarter, 2008, as was the capability for sites to administer their own circuits.

![Figure 14. OSCARS generated and managed virtual circuits at FNAL – one of the US LHC Tier 1 data centers. This circuit map (minus the callouts that explain the diagram) is automatically generated by an OSCARS tool and assists the connected sites with keeping track of what circuits exist and where they terminate.](image)

**Inter-domain collaborative efforts are extensive and well developed**

Efforts to ensure that the virtual circuit service will operate across network domains (e.g. between ESnet, Internet2, and GÉANT) are extensive and well developed.

Terapaths developed at BNL and LambdaStation developed by FNAL are both systems designed to provide guaranteed bandwidth within the site / Lab domains. OSCARS has been designed to interoperate with these existing systems and inter-domain interoperability for layer 3 virtual circuits was demonstrated third quarter 2006 (Terapaths) and for layer 2 virtual circuits demonstrated at SC07 (fourth quarter 2007) for both Terapaths and LambdaStation.

Seamlessly setting-up optical circuits across independently operated networks is the goal that will make the virtual circuit service useful to the science community and this requires the coordination of multiple administrative domains. This is achieved through compatible control plane software enabling provisioning across domain boundaries with the appropriate authentication and authorization. Compatible control plane software is under development through several on-going projects, including the
The NSF-funded DRAGON project, the ESnet OSCARS program, and the GÉANT2 AutoBAHN project. Interoperability of circuits set up across these domains was demonstrated at SC07 (fourth quarter 2007).

The DICE working group (DANTE/GEANT, Internet2, Caltech, ESnet) coordinates the work and issue resolution (both software and operational) for cross-domain virtual circuits. Recent DICE accomplishments include a draft of topology exchange schema being formalized in collaboration with OGF’s Network Measurement Working Group (NMWG), and interoperability was demonstrated 3Q07. (This capability is related to discovering available paths through networks outside of your own domain. Likewise, an initial implementation of cross-domain reservation and signaling messages (the DICE InterDomain Controller Protocol (IDCP)) demonstrated at SC07. Other multi-organization activity includes interoperability testing with the Nortel (Canadian telecom equipment manufacturer) DRAC (Dynamic Resource Allocation Controller) which implemented the IDCP on their platforms.

**OSCARS is a production service in ESnet**

OSCARS is now deployed as a prototype production service in ESnet and has the status of all other ESnet services. It is, however, a very new service and still evolving.

As a production service, OSCARS is undergoing a continual reassessment and is adapting to user experiences with the service.

User experience in the first year of OSCARS operation has revealed several new capabilities that are required. In spite of the fact that one of OSCARS primary goals was to provide hard bandwidth guarantees, it has turned out that this needed to be relaxed. It is useful to permit over-subscribing a path in order to accommodate backup circuits and to allow for site managed load balancing.

Another area that has become clear is that there is a need to direct routed IP traffic onto SDN in a way transparent to the user. There are many issues here and this is an area of research and experimentation at the moment.

**2.2.5. Planning Process 4. Develop outreach approaches that identify what communities are not being well served by the network because they are not equipped to make effective use of the network**

Assistance and services are needed for smaller user communities that have significant difficulties using the network for bulk data transfer. This issue cuts across several SC Science Offices, and these problems must be solved if a broad category of scientists are to effectively analyze the data sets produced by petascale machines. *Addressing these problems is essential.*

Persistent performance problems exist throughout the DOE Office of Science (and everywhere else in the science community except a few large collaborations like those associated with the LHC). Existing tools and technologies (e.g. GridFTP, TCP tuning) are not deployed on end systems or are inconsistently deployed across major resources. The resulting performance problems impede scientific productivity – unreliable data transfers soak up scientists’ time, because, e.g., they must babysit transfers.

Default system configuration is inadequate and most system administrators don’t know how to properly configure a computer for WAN data transfer. Scientists and system administrators typically don’t know that WAN data transfer can be high performance, so they don’t ask for help.
Tools and technologies for high performance WAN data transfer exist today. TCP tuning documentation exists, and tools such as GridFTP are available and are used by sophisticated users. DOE has made significant contribution to these tools over the years and one of the things that ESnet has done is to set up a web site devoted to information / best practice on bulk data transfer, host tuning, etc. (fasterdata.es.net)

Sophisticated users and programs are able to get high performance – user groups with the size and resources to “do it themselves” get good performance (e.g. HEP, NP). Smaller groups do not have the internal staff and expertise to manage their own data transfer infrastructures, and so get low performance. The WAN is the same in the high and low performance cases but the end system configurations are different.

Apart from tuning and WAN tool deployment, other potential approaches include various latency hiding forwarding devices in the network which is a current topic of R&D and experimentation in ESnet and in Internet2 (see subsection “Transparent Acceleration of Data Transfers”, below).

As a result of the experience ESnet has gained in the requirements workshops a path forward has been identified (which would probably not be a lot different for any similar set of users, e.g. in university environments):

- Task one entity (within the SC community) with development, support and advocacy for WAN data transfer software
- Port GridFTP to new architectures – we need these tools to work on petascale machines and next-generation data transfer hosts
- Increase the usability of the tools – scientific productivity must be the goal of these tools, so they must be made user-friendly so scientists can be scientists instead of working on data transfers
- Work toward a consistent deployment – all major DOE facilities must deploy a common, interoperable, reliable data transfer toolkit (NERSC, ORNL, light sources, nanocenters, etc)
- Workflow engines are needed to automate the transfer of large data sets or large numbers of files
- Test and monitoring infrastructure (e.g. iperf+PerfSONAR) must be available at every site

However, several of these have the caveat that middleware must have a long term support infrastructure or scientists will be leery of becoming dependent on something that might become unsupported in the future.

These problems MUST be solved if scientists are to effectively analyze the data sets produced by petascale machines.

2.3. Strategy III: Anticipating future network capabilities that will meet future science requirements with an active program of RESEARCH and Advanced Development

The next generation networks and services that are needed to support the new science environment are being defined by examining three types of requirements: the data generating and use characteristics of new instruments and facilities; changes in the process of doing science in this new environment, and examination of the future implications of historical trends in network traffic.

Additionally, advances in the state of telecommunications and network technology over the past several years enable revolutionary approaches to providing national-scale
network services. Concurrently, middleware and applications architectures are increasingly supporting wide area distributed systems, introducing fundamental changes in the nature and profile of network traffic as observed by network providers. Hybrid network architectures - involving both traditional Internet connectivity and scheduled, targeted (circuit-like) capacity - offer the potential to provide a radically different set of interaction modes between the network and applications and middleware.

To move beyond proof-of-concept demonstrations and initial deployment for sophisticated users toward persistent, reliable services it will be necessary to harvest the best concepts shown to be feasible and systematically move them into production services. To support this new set of services, and these new application and middleware systems, there must also be deep understanding of the behavior of the systems and the interactions between applications, middleware, and network services.

In April, 2007 the DOE Office of Science organized a workshop to bring together a small group of experts to work with the ESnet team to examine the current roadmap, roadmaps of similar enterprises, user requirements, and new technology options. The objective of the workshop was to create a new, multi-year technology roadmap for ESnet that identifies milestones and partners. The document from which this section is abstracted‡‡ is an updated and condensed version of the report from that workshop. The workshop report is available at www.es.net/hypertext/ESnetRD-Workshop-Report.pdf and a prioritized list of tasks is available in the appendix of that report.

These topics include:

- Guaranteed network bandwidth
- End-to-end monitoring
- Transparent acceleration of data transfers
- Federated Trust

3. Summary

ESnet and the Office of Science have put in considerable effort in working with the SC community to identify the networking implication of the instruments, supercomputers, and the evolving process of how science is done.

The resulting understanding of the current and evolving needs of the SC community has resulted in a new network environment that we believe will enable the distributed aspects of SC science. The new network and its services are not a static implementation, but continuously evolving as the reassessment of the effectiveness of the original plans indicate needed changes and because the ways in which the science community uses the network changes with an evolving process of science.

In order to avoid limiting the scope of solutions to engineering practices that are developed and understood, ESnet and SC use an active program of research and advanced development effort to keep anticipating future network capabilities that will meet future science requirements.

An extended version of this paper is available at www.es.net/pub/esnet-doc.

‡‡ This section is abstracted from “ESnet Research Priorities, August 2008” which is available at http://www.es.net/hypertext/Adv-Dev-Projects.html
Acknowledgements

In addition to the authors, the ESnet senior network engineering staff that is responsible for the evolution of ESnet also contributed many ideas: Joseph H. Burrescia, Michael S. Collins, Jon Dugan, James V. Gagliardi, Yvonne Y. Hines, Kevin Oberman, and Michael P. O’Connor. Steve Cotter is now the head of ESnet. ESnet is funded by the US Dept. of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) program. Dan Hitchcock was the ESnet Program Manager when ESnet4 was being planned. Vince Dattoria is the current Program Manager. Thomas Ndousse-Fetter is the Program Manager for the network research program that funded the OSCARS project. ESnet is operated by Lawrence Berkeley National Laboratory, which is operated by the University of California for the US Dept. of Energy. This work was supported by the Director, Office of Science, Office of Advanced Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[4] perfSONAR is an infrastructure for network performance monitoring, making it easier to solve end-to-end performance problems on paths crossing several networks. It contains a set of services delivering performance measurements in a federated environment. These services act as an intermediate layer, between the performance measurement tools and the diagnostic or visualization applications. See www.perfsonar.net
The European Grid Initiative and the HPC Ecosystem

Per ÖSTER
CSC - IT Center for Science Ltd., Finland

Abstract. Through national initiatives and projects funded by the European Commission a number of e-Infrastructures have evolved in Europe. Over a number of years different grids have matured from test and proof-of-concept to full-fledged production infrastructures with large-scale research collaborations dependent on their existence. In this article the European Grid Initiative Design Study and the vision to create a sustainable organisation for European e-Infrastructure and its role in the HPC ecosystem is presented and discussed. The EGI Blueprint is presented and found to be complementary to the initiative for extreme scale computer facilities in Europe represented by the Partnership for Advance Computing in Europe.

Keywords. Grid computing, High-Performance Computing, e-Infrastructure, e-science.

Introduction

Grid computing [1] is a recently developed technology for access and sharing of distributed storage, computing resources, sensors, and other networked resources. Over the past decade, European states and the European Commission have invested heavily in grid technologies and e-Infrastructures in general and Europe has reached a leading position in grid development and deployment. Moreover, Europe has also been highly active in accelerating the corporate and societal usage of grids. The European grid infrastructure supports a wide range of applications, which are developed from test bed projects, to efforts in infrastructure deployment and management. In a number of European countries, national grid projects and related e-Science projects have been initiated.

Today, many research communities and application domains already rely on grids. The present challenge is to satisfy the need to protect the existing services and the investments made by both the user communities and funding organizations. The cyclic based EU project funding for e-Infrastructures has been crucial in reaching the present high level of grid developments. However, it is not enough anymore, and if Europe wants to maintain its leading position on the global science arena, there is an urgent need to ensure a reliable and adaptive support for utilization in all research areas of these e-infrastructures. Today’s grid users range from grid enthusiasts to dependent large research collaborations, and in that sense, tomorrow’s grid users are already here

1 Corresponding Author: Per Öster; E-mail: per.oster@csc.fi.
asking for a long-term perspective. Therefore, the creation of a sustainable grid infrastructure is the logical next step in the development.

In this article we describe how the need of existing project based European e-Infrastructures to evolve into a sustainable model is addressed by the European Grid Initiative Design Study (EGI_DS) [2] and how the model of a European Grid Initiative (EGI) [3] relates to the general computational science landscape or HPC ecosystem in Europe.

1. Development steps towards a sustainable grid infrastructure

High-speed networking may be considered as one of the first crucial factors in the rate of progress in grid computing. In Europe, the high speed networking infrastructure was developed with substantial support by the European Commission’s 5th Framework Programme funded GÉANT project [4], which started in 2000 and which provided the underlying network infrastructure for the early Grid projects.

The success of GÉANT brought more funding for grid projects that were started under the 6th Framework Programme of the European Commission. The largest project and most prominent example is the Enabling Grids for E-sciencE (EGEE) [5] project, which was launched in 2004 with the objective to construct a multi-science Grid infrastructure for the European Research Area. The project linked the national, regional and thematic Grid efforts, and was interoperated with other Grids around Europe and the rest of the globe. The resulting high capacity of this worldwide infrastructure greatly surpassed the capacities of local clusters and individual centres and provided a unique tool for a collaborated computer-intensive science, e-Science. This, in turn, directed the work efforts towards interoperability and towards European or even global Grid infrastructures. The EGEE project was followed by the EGEE II project in 2006, and subsequently by the EGEE III project, which was launched with the funding of the 7th Framework Programme in May 2008.

The idea of sustainability is supported widely within the whole grid community. The e-Infrastructure Reflection Group (e-IRG) [6] published a recommendation [7] in December 2005 stating the following: “The e-IRG recognizes that the current project-based financing model of grids (e.g., EGEE (Enabling Grids for E-science), Distributed European Infrastructure for Supercomputing Applications (DEISA) [8] presents continuity and interoperability problems, and that new financing and governance models need to be explored – taking into account the role of national grid initiatives as recommended in the Luxembourg e-IRG meeting.” The sustainability has also been strongly addressed by the European Commission. Viviane Reding, Commissioner of the European Commission, stated at the EGEE’06 Conference that, “for Grids we would like to see the move towards long-term sustainable initiatives less dependent upon EU-funded project cycles”.

This can be seen in the funding trends of the European Commission’s different programmes. The 5th Framework programme concentrated on broad scale test-beds while the 6th Framework Programme encouraged production quality facilities and infrastructures. The e-Infrastructures established during FP6 represent a big step forward in Europe. The efforts required to build a large-scale production e-Infrastructure are vast and in order to protect Europe's investment, the next phase of the pan-European e-Infrastructure needed to be based on existing e-Infrastructures. In order to ensure the long-term sustainability, shared e-Infrastructures should be economically
and resource-wise independent of any individual user community or resource provider. The 7th Framework programme opens a new page by supporting the sustainable grid- and data-based e-Infrastructures. This should also foster emergence of new organisational models to consolidate a sustainable approach to e-Infrastructures, in particular in the domain of grids and data repositories. New service provisioning schemes should also be more neutral and open to all user communities and resource providers.

2. European Grid Initiative Design Study

The European Grid Initiative Design Study (EGI_DS) project was launched in September 2007 with a lifetime of 27 months. The project is funded through the European Commission’s 7th Framework Programme. The goal of EGI_DS is to create the conceptual setup and operation of a new organizational model of a sustainable pan-European grid infrastructure. The project evaluates use cases for the applicability of a coordinated effort, identifies processes and mechanisms for establishing the EGI, defines the structure of a corresponding body and initiates the construction of the EGI. At its start, the EGI_DS was already supported by 38 National Grid Initiatives (NGI). The project will last until the end of November 2009, and according to the current plans, the EGI organization will be operational at the beginning of 2010. The timetable, Figure 1, is critical with several of the ongoing e-Infrastructure projects already ending during the first half of 2010.

![Figure 1: Timetable for the EGI Design Study and the EGI Organization.](image)

The specific objectives set for the EGI Design Study are to:

1. Consolidate the requirements for an EGI from the member states and for as broad range of academic disciplines as possible.
2. Define the functional role of the EGI with respect to the NGIs at the start of the EGI, and plan for the evolution of its functions as it matures.
3. Establish a legal, financial and organizational basis for the EGI to undertake these functions that has support from the member states.
4. Establish the EGI organization and manage the transition from the existing project based grid operation and support to coordination of sustainable production services.
5. Ensure that all stakeholders within the member states, international standards bodies, research grid services in other countries, etc are aware of the EGI and have manageable relationships with it.

EGI_DS has nine partners forming the core of the “EGI Preparation Team” leading the transition to a sustainable European grid e-Infrastructure.

The project has initiated the creation of the EGI Policy Board consisting of representatives of NGIs and observers from other e-infrastructure projects (DEISA [8], EGEE [5], GÉANT [4] and PRACE [9]). The EGI_DS project team submits a sequence of proposals for endorsement, key to the establishment of the EGI to the EGI Policy Board. The Policy Board is an essential element for interaction between NGIs and experts from the existing infrastructure projects to determine which parts of the operation will be performed by the EGI organization, and which parts will remain within the responsibility of the NGIs. It also identifies the level of maturity and mandate required from the individual NGIs in order to provide steering and control of the operation of the infrastructure.

2.1. Vision and objectives of the EGI

The European Grid Initiative will enable the next leap in research infrastructures by defining and installing the EGI organization (EGI.org) for the support of all sciences in the European Research Area, as defined in the EGI Vision Paper [10]. The National Grid Initiatives will form the core of the EGI. The EGI organization will coordinate and operate a common multi-national, multi-disciplinary and, multi-technology Grid infrastructure. The organization will enable and support international grid-based collaboration, provide support and added value to NGIs and liaise with corresponding infrastructures outside Europe.

To ensure that Europe fully capitalises on its large investments in Grid infrastructures, middleware development and applications, the objectives of the future EGI organization were set to be to:

- Ensure the long-term sustainability of the European e-infrastructure;
- Coordinate the integration and interaction between National Grid Infrastructures;
- Operate the European level of the production grid infrastructure to link National Grid Infrastructures for a wide range of scientific disciplines;
- Provide global services and support that complement and/or coordinate national services;
- (Authentication, VO-support, security, etc);
- Coordinate middleware development and standardization to enhance the infrastructure by soliciting targeted developments from leading EU and national grid middleware development projects;
- Advice national and European funding agencies in establishing their programmes for future software developments based on agreed user needs and development standards;
- Integrate, test, validate and package software from leading grid middleware development projects and make it widely available;
• Provide documentation and training material for the middleware and operations. (NGIs may wish to make the material available in turn in their local language);
• Take into account developments made by national e-science projects which were aimed at supporting diverse communities;
• Link the European infrastructure with similar infrastructures elsewhere;
• Promote grid interface standards based on practical experience gained from grid operations and middleware integration activities, in consultation with relevant standards organizations;
• Collaborate closely with industry such as technology and service providers, as well as grid users, to promote the rapid and successful uptake of Grid technology by European industry.

2.2. The EGI Blueprint

The grid infrastructures, now operational over many European countries, were established through a series of projects starting from early 2001 with the European Data Grid (EDG) under the fifth Framework Program. After this proof-of-concept, which demonstrated the potential impact of Grid technologies and infrastructures on European science, the first large-scale production grid was deployed by the Enabling Grids for e-Science project (EGEE). Its second and third phase, EGEE-II and EGEE-III, the latter ongoing until first quarter 2010, have provided a large-scale production quality grid for European scientists and their collaborating partners all over the globe. In parallel to this major effort, many regional or scientific grid activities and projects contributed to the Grid infrastructure ecosystem are available in Europe today.

Though the major driving force behind EGEE was the High Energy Physics community with the need of analysing enormous amounts of data from the Large Hadron Collider (LHC), many other disciplines contributes to and exploit the European Grid infrastructure.

EGEE and similar European Grid infrastructure projects rest on three pillars for financial support:

• Co-funding by the European Commission (EC)
• National funding matching EC funding
• Funding of computing infrastructures, their operation and maintenance by national/institutional sources

The EGI Blueprint [11] describes the proposal developed by the EGI Design Study to establish a sustainable grid infrastructure in Europe to be in place by the end of EGEE-III, that is, spring 2010. The Blueprint is based on the vision of a large pan-European distributed computing and data grid infrastructure providing such services as described in the EGI Vision Paper [10].

The EGI Blueprint is a proposal on how to realise such a vision, with the necessary implications for the implementation, operation, user interaction and management of the corresponding infrastructure thereby supporting collaborative scientific discoveries in the European Research Area.

Although all effort must be made to ensure continuity for the current users of European infrastructure projects, especially, but not limiting to EGEE, the EGI is not a
simple continuation of EGEE. Most existing infrastructure projects have made direct
agreements with resource providers; in contrast, the EGI concept is built on each
member state’s establishment of its own National Grid Initiative (NGI) which can
represent the resources in its country and which can interface to the EGI.

It is important to note that the EGI is based on the NGIs and that EGI.org should
be understood as the “glue” enabling coherence between the NGIs for the benefit of
their user communities. EGI.org will link existing NGIs and will actively support the
setup and initiation of new NGIs that do not yet exist. The relationship between
EGI.org and the NGIs is governed by the “subsidiary principle” which means that tasks
that are more effectively performed at the national or regional level should be left there.
EGI.org will ensure a pan-European Grid coordination – with aims to standardize
wherever reasonable.

EGI.org will provide central functions to address the operation of the
infrastructure, user support and application development, middleware interfaces and
final certification, and management. It should be noted that the resources needed for
this coordination operation represent only a small fraction, equivalent to a few percent,
of the total effort spent on Grids in Europe today.

It is necessary that the underlying middleware for the European grid can be
developed continuously. It is proposed in the Blueprint that this development continues
to take place in the “middleware consortia” and not be part of the EGI funding model.
But, EGI.org will foster middleware commonality and interoperability wherever
possible. The picture is complex with several different middleware stacks and toolkits
deployed in Europe such as, ARC [12], gLite [13], Globus [14], and UNICORE [15].
Interoperation and interoperability will be major issues for EGI and to address this, a
unified middleware distribution (UMD) has been proposed by the development
communities around ARC, gLite and UNICORE.

For the successful launch of EGI.org an initial period with co-funding from the
European Commission could be found necessary. The major purpose of this co-funding
would be to bring all the stakeholders — NGIs — together, but not to substitute for the
national funding which will be the base of EGI financial stability and sustainability.
National funding will have to play an increasing role in the EGI even though the cost to
operate the grid should stabilize as the infrastructure expands, thanks to streamlining
and automation. In the highly dynamic environment of distributed computing for
science, funding for innovation has to be maintained – most logically on a project
basis.

2.3. Role of the National Grid Initiatives in EGI

Several European countries have launched, or are about to launch, National Grid
Initiatives (NGI). NGIs are planned to be the building blocks of the new sustainable
grid infrastructure. As the main actors of EGI, NGIs ensure the operations of the grid
infrastructures in each country and a transparent representation of the requirements of
all their scientific communities together with resource provider and all e-Infrastructure
related Institutions. The goal of EGI is to link existing NGIs together and to actively
support the setup and initiation of new NGIs in those countries where corresponding
efforts do not yet exist. The characteristics of the NGIs can be identified as follows:
Each NGI should:

- Be the only recognized national body in a country with a single point-of-contact representing all institutions and research communities related to a national grid infrastructure.
- Have the capacity to sign the Statutes of EGI.org – either directly or through a legal entity representing it.
- Have a sustainable structure or be represented by a legal entity having a sustainable structure in order to commit to EGI.org in the long term.
- Mobilize national funding and resources and be able to commit to EGI.org financially i.e. to pay the EGI.org membership fee and – if there is a demand for such services in the NGI – order EGI.org services and pay for these services.
- Ensure the operation of a national e-Infrastructure to an agreed level of service and its integration in EGI.
- Support user communities (application independent, and open to new user communities and resource providers).
- Contribute and adhere to international standards and EGI policies and quality criteria.

At this point in time, NGIs in Europe are at different stages of their implementation, ranging from individuals representing an emerging NGI to established NGIs with a legal status. During the development of EGI, early forms of NGIs are expected to transform into legal entities, which are able to collaborate on formal grounds in the European e-Infrastructure landscape. The EGI effort intends to support this development in order to achieve a mature level for all NGIs in Europe.

NGIs are of course fully autonomous in their organization but must accept the boundary condition of the EGI model and the established EGI policies and rules. The autonomy of the NGIs is not limited to the tasks each NGI performs for supporting the national grid infrastructure, its users, and research communities, but naturally extends also to the tasks that are specifically aimed at allowing the sharing of the national IT resources at pan-European and international level in a uniform, robust and seamless way and at supporting international application communities. The EGI model and the EGI.org coordination intend to promote such autonomy and related responsibilities, proposing only the specifications, operational rules and policies needed for maintaining a good working infrastructure and for the benefit of users, applications and resource providers.

During the EGI Design Study, an EGI Policy Board has been formed by NGI representatives and serves as the essential body for interaction between EGI_DS and the NGIs and their user communities in the respective participating countries.

The EGI_DS has created a wiki-based Knowledge Base that is used in collaboration and communication with the NGIs. The EGI Knowledge Base (http://knowledge.eu-egi.eu) provides up-to-date information on National Grid Initiatives, use cases, as well as increasingly detailed plans for the future European Grid Infrastructure. NGI Representatives have direct access to their page, and are asked to provide their input and update their local information in the Knowledge base. The Knowledge-base is intended to serve not only the development within the EGI_DS project, but also to be an aid and source of information for NGI representatives in their interaction with national funding agencies and policy making bodies.
2.4. Resource Providers and EGI

Within the NGIs the NGI itself, or other resource providers and funding agencies, collectively fund, procure, host and operate the IT resources for the research application areas, typically in specific resource centres. EGI is not primarily intended to act as a direct resource provider; thus EGI and EGI.org should not be expected to impose any changes in the relationship between research application areas and funding agencies. The NGI’s services include however, such general services as global monitoring and accounting tools that are supposed to be useful, if not necessary for resource providers and owners in order to verify and tune the resource allocation to the research areas it supports and the overall resource usage. In the same way EGI should not be expected to change anything in the present funding models for the resource centres.

EGI is intended to strongly interact with the resource centres, acting as their entry point into the global grid, and playing a facilitator role for centres of different sizes, serving different research application areas, with different requirements, without imposing any unnecessary hierarchy. The resource centres hardware is not, per se, part of the EGI infrastructure, as defined above, but a resource centre (with its hardware and the usual system level support for it and for the applications deployed) joins the EGI grid via its reference NGI, which, being part of the EGI, takes care of making its resources part of the global grid. The resource owners will of course still be free to decide on the mechanisms for deciding who has the right to use their resources.
The resource centres are included in the EGI organization only through their reference NGI, but are the final destination of much of the activity included in the operation and security function of EGI.org. As the resource centres are responsible for the operation and the security of the IT resources and Grid services they host, they have to apply needed corrective actions when EGI.org detects any kind of malfunction affecting the grid.

2.5. Middleware Consortia and EGI

The development of middleware will not be a direct responsibility of the EGI. However, a function within EGI.org that has a close relation with different middleware development teams will be established and will have two main tasks:

1. Guarantee the maintenance and verify the interoperation or interoperability of the middleware currently deployed on the e-Infrastructure – this is a task similar to any software maintenance, with its rules and costs provided as a general service offered to all European grid users.
2. Coordinate and communicate to the middleware developers further developments requested by the research communities, virtual organizations and operational teams.

Even if these tasks are strongly related they are conceptually different and should be treated separately in terms of financial responsibility.

To guarantee the continuity of the grid infrastructures in Europe, natural partners for EGI are the middleware consortia in Europe with the most widely deployed middleware stacks: gLite, ARC, and UNICORE. The US Virtual Data Toolkit (VDT) with Globus and Condor also plays an important role in Europe and a relationship with these developers must also be maintained.

EGI would most likely support the evolution of the middleware in the form of an EGI Unified Middleware Distribution (UMD), which does not constitute the development of a new middleware stack but implements a unified distribution of certified components of the stacks which are currently deployed in the European e-Infrastructure. In such a model, EGI.org will include only a small team for the final certification of the accepted components.

2.6. The Users of the Grid and EGI

The ultimate purpose of EGI and its Grid Infrastructure is to provide a research infrastructure to the benefit of European research communities. The specific value EGI offers to the research application areas is an infrastructure enabling them to share and access, in a secure and transparent way, IT resources, data and scientific instrumentation provided by resource centres globally.

During the evolution of grid projects in the European Union Framework Programs (FP5-FP7) the number of user communities and research applications have been steadily growing. Although High-Energy Physics and the demanding computing requirements of the Large Hadron Collider (LHC) is the major user community of European grids today, the growing number of scientific disciplines in EGEE alone adds at least 200 Virtual Organizations (VOs) with more than 7500 users relying on production quality grids and the underlying infrastructure. The research application
areas vary greatly and the list is continuously growing. Currently it includes areas such as:

- Archaeology,
- Astronomy & Astrophysics,
- Civil Protection,
- Computational Chemistry,
- Computational Fluid Dynamics,
- Computer Science/Tools,
- Condensed Matter Physics,
- Earth Sciences,
- Finance (through the Industry Task Force),
- Fusion,
- Geophysics,
- High-Energy Physics,
- Life Sciences,
- Multimedia, and
- Material Sciences.

These or similar sets of research areas are also supported within other projects such as BalticGrid-II [16], DEISA [8], EELA-2 [17] and SEE-GRID-Sci [18], to mention a few. It is also very important that these research areas, which use various current grid infrastructures supported by EU-, nationally-, and regionally funded projects can make a transition without disruption to the envisaged EGI/NGI-based model.

Users of an infrastructure need support and EGI.org is foreseen to at least have a coordinating function, ensuring collaboration between support efforts within the different NGIs.

3. European Grid Infrastructure and Large Scale Research

The European Strategy Forum on Research Infrastructures (ESFRI) has 35 different research infrastructures on its roadmap. In the present update procedure (year 2007-2008) this number will grow even further. The newly proposed infrastructures alone have a construction cost of the order of 2 Billion Euro.

Many of the proposed infrastructures are in a preparatory phase, planning their construction. The need for Information and Communication Technologies (ICT) functions is investigated. Common keywords are data management, computing and simulation, grids, authentication and authorisation, network capacity, and software development. These needs are common for many of the projects. This has also been recognised by the European e-Infrastructure Reflection Group (e-IRG). The group emphasised in a recent report [19] to the European Commission the need for horizontal ICT functions common to the ESFRI projects. Horizontal functions (Figure 3) are often referred to as e-Infrastructure for e-Science.

An e-Infrastructure has entered the ESFRI roadmap in the form of PRACE [9] and High-Performance Computing facilities. This area has previously been left for the USA and Japan. But, with today’s research challenges within areas such as climate research, biomedicine, and the fundamental sciences, the need for leading high-performance computing facilities has been recognised by the European Commission and a number of countries leading to the EU project, Partnership for Advanced Computing in Europe (PRACE). More ICT related projects might enter the list and, as an example, for the 2008 update of the ESFRI list two new proposals for horizontal ICT infrastructures or e-Infrastructures has been proposed by Finland, “Infrastructure for Preservation of
Unrevealed Scientific Data” (IPURE) and “European Software Services Network for Large-Scale research Facilities” (ESSN). Both these proposals are answers to a general need by large-scale research projects for timely and reliable e-Infrastructures.

3.1. Grids and the HPC Landscape

A balanced e-Infrastructure is needed to meet the growing needs of compute- and data-intensive research. Such infrastructure have often been mapped as a tiered pyramid structure [20] to depict the scaling in computing capacity or number of installations between the different layers of resources, covering everything from personal computers to supercomputers. In this layered structure the researcher usually utilizes resources in several different layers. The personal computer is used for the daily work and simpler pre- and post-processing, departmental or university resources are used for the regular computing tasks and regional, national or international resources are used for the peer reviewed most demanding computing tasks. In this structure grid technologies play the role of an integrating technology, allowing efficient and dependable access to and sharing of different kind of resources across organizational borders. The qualities of grid technology applies equally well to all of the different elements of the HPC ecosystem, from local resources to the very largest super computers.

In this perspective, it is evident that the collaboration between the EGI and HPC initiatives like PRACE [9] is of vital importance for a successful and efficient European HPC infrastructure.
4. Conclusion

Grids and grid technology is a key necessity for large-scale research. Many research communities are already dependent on grids and expect that today’s production grids like BalticGrid, DEISA, EGEE, SEE-Grid-Sci and many more will continue and develop with a long-term perspective to become a persistent research infrastructure in the same way as the European research network (GEANT). Of the user communities, the High-energy physics community with its Large Hadron Collider (LHC) experiment has, for example, based the whole simulation and analysis of experimental data on a worldwide network of computing and storage resources. This is an infrastructure operated by several grid projects where EGEE and OSG (in the USA) are the largest. The proposed EGI organisation is planned to cater for LHC and other user communities’ need for a coordinated operation of all grid enabled resources in the European countries.

The PRACE project aims at adding to the European HPC ecosystem by giving European researchers access to the very largest kind of computing resources. With the establishment of PRACE, European researchers will gain access to resources of a capability that previously have been available only in the USA and Japan. This will be an important step for computational science in Europe and can be foreseen to stimulate the whole area of computational science and its application.

EGI and PRACE are addressing different needs within computational science: EGI the organization and sharing of different types of computing resources and PRACE the access to the very largest resources. The two initiatives can be considered complementary and will together form the base for an accelerating development of computational science in Europe.

Acknowledgement

This paper was produced partly as an activity within the FP7 EU Project European Grid Initiative Design Study and benefits from material in form of texts and illustrations produced within the project by the different EGI_DS project partners, Table 1. Special thanks to Katja Rauhansalo (CSC) who has contributed substantially gathering material and writing the first drafts of this paper.

Table 1: EGI Design Study Partner Organizations

| 1. | CERN – European Organization for Nuclear Research, Switzerland |
| 2. | CESNET – CESNET, z.s.p.o., Czech Republic |
| 3. | CNRS – Centre National de la Recherche Scientifique, France |
| 4. | CSC – IT Center for Science Ltd, Finland |
| 5. | DFN – Verein zur Foerderung eines Deutchen Forschungsnetzes – DFN – Verein, Germany |
| 6. | GRNET – Greek Research and Technology Network S.A |
| 7. | GUP – Johannes Kepler Universität Linz, Austria (in 2008 replaced by LMU – Ludwig-Maximilians Universität München, Germany) |
| 8. | INFN – Instituto Nazionale di Fisica Nucleare, Italy |
| 9. | STFC – Science & Technology Facilities Council, UK |
References


[4] EU FP5 Project GEANT (multi-gigabit pan-European data communications network and project), URL: http://www.geant.net/


[8] EU FP7 Project Distributed European Infrastructure for Supercomputing Applications (DEISA), URL: http://www.deisa.eu/

[9] EU FP7 Project Partnership for Advanced Computing in Europe (PRACE), URL: http://www.prace-project.eu/


[16] EU FP7 Project BalticGrid Second Phase (BalticGrid-II), URL: http://www.balticgrid.org/


Grid and e-Science in Korea

Kihyeon CHOA,1

“High Energy Physics Team, KISTI, Daejeon 305-806, Korea

Abstract. We present the history and current status of grid and e-Science activities in Korea. Until 2006, we focused on research on grid technology and infrastructure. Since 2007, we have been enabling e-Science in four major areas: physics, life science, engineering, and tele-science. We have also provided the international and domestic research communities with both infrastructure and technology. We present the current status and activities of grid and e-Science projects in Korea. As an example of e-Science, we present the concept and deployment of e-Science for high-energy physics.

Keywords. Grid, e-Science, High Energy Physics and Data Processing

Introduction

We report our experiences and results relating to the utilization of cyberinfrastructure in various fields of science in Korea. According to the Wiki webpage on “cyberinfrastructure,” the term “cyberinfrastructure” describes the new research environments that support advanced data acquisition, storage, management, integration, mining, and visualization, as well as other computing and information processing services over the Internet [1]. In the scientific sense, cyberinfrastructure is a technological solution to the problem of efficiently connecting data, computers, and people with the goal of enabling the development of novel scientific theories and knowledge [1].

Cyberinfrastructure consists of resources, middleware, and user environments. Resources refer to networks and computers. Middleware represents grids for resource allocation, information service, security, data management, and monitoring. Applications of e-Science include physics, engineering, life science, and tele-science.

In Korea, cyberinfrastructure has been developed through the growth of resources, grids, and e-Science projects. We report on the resources, grids, and e-Science projects in Korea [2]. We also report on global collaborations for these projects.

1. Cyberinfrastrturcure Resources

1.1. Overview

KISTI (Korea Institute of Science and Technology Information) is a government-aided research institute that provides Korean scientists and engineers with scientific and
industrial databases, research networks, and supercomputing infrastructure. The mission of KISTI is to advance the national information infrastructure by providing the communities involved in advanced research with leading-edge computational resources and networks. The supercomputing center at KISTI began as the KIST (Korea Institute of Science and Technology) computer laboratory in 1967, and in 1988, it began to provide supercomputer resources. Over the last few decades, the center has grown to provide high-end supercomputing and networking technologies, along with active support and services, to various groups of research communities in universities and industries. The center also collaborates and builds partnerships with various scientific communities both nationwide and worldwide. The overall goal of these efforts is to foster state-of-the-art research and overcome scientific challenges by applying the most innovative and developed technologies.

KISTI also supports grid and e-Science projects in Korea. Therefore, KISTI is known as the national headquarters for supercomputer, network, grid, and e-Science projects.

1.2. High-performance Computing

The goal of the KISTI supercomputing center is to serve as a national center that enables scientists and engineers to achieve their R&D goals. The performance-enhancing facilities available are world-class supercomputing infrastructure, networks, and technology. We provide various types of supercomputers to help meet the demands of scientific and engineering activities performed in universities and industries. High-performance computational platforms provide researchers with a wide range of cutting-edge equipment. We operate powerful high-end computing resources.

The center has various programs designed to support application research, as part of its aim to promote the advancement of computational science. We support research in various scientific and engineering disciplines. We conduct user support programs, form research groups, and provide technical support to users, including those affiliated with education and consulting. We also develop advanced supercomputing techniques for the effective use of computational resources.

Currently, we are installing the fourth supercomputer, which will help achieve a computing performance of 322.6 TFlops in various user support programs; this places the center among the top ten providers of computing facilities in the world. Table 1 lists the specifications of the fourth supercomputer, which will be installed by 2009.

<table>
<thead>
<tr>
<th>Item</th>
<th>SMP System</th>
<th>Multiprocessing</th>
<th>Cluster system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Phase I</td>
<td>Phase II</td>
<td>Phase I</td>
</tr>
<tr>
<td>Manufacturer &amp; Model</td>
<td>IBM p 595</td>
<td>IBM p6H</td>
<td>SUN 6048</td>
</tr>
<tr>
<td>Processor</td>
<td>POWER5+ 2.3</td>
<td>POWER6 5GHz+</td>
<td>AMD Opteron 2GHz (Barcelona)</td>
</tr>
</tbody>
</table>

Table 1. Specifications of the fourth supercomputer at KISTI.
1.3. High-performance Networking

The next topic is networks. We have also implemented KREONET (Korea Research Environment Open NETwork), which is a national R&D network supported by MEST (Ministry of Education, Science and Technology), Korea, since 1988. It provides Korean R&D communities with high-performance network services in order to enable the exchange of information on science and technology. KREONET comprises 15 regional network centers, including KNU (Kyungpook National University), GIST (Gwangju Institute of Science and Technology), and Jeju National University. By using the regional network centers, we establish a network infrastructure that supports a range of location-independent R&D activity. It provides an important national R&D network infrastructure for cooperative state-of-the-art R&D activity [3].

In addition, in order to boost advanced applied R&D in Daejeon Science Town, KREONET provides a specialized 10-Gbps backbone in the form of the superSIREN (Daduk advanced science and technology research network) that links seven organizations: KISTI, KAIST (Korea Advanced Institute of Science and Technology), KBSI (Korea Basic Science Institute), KAERI (Korea Atomic Energy Research Institute), KRB (Korea Research Institute of Bioscience and Biotechnology), KIGAM (Korea Institute of Geo science And Mineral resources), and CNU (Chungnam National University). Approximately 50 institutes are connected with >1 Gbps high-bandwidth links, as part of an advanced R&D network for state-of-the-art application R&D [3].

It is also important to note that KREONET is connected to GLORIAD (Global Ring Network for Advanced Applications Development). As shown in Figure 1, GLORIAD connects locations across the world with a 2.5–10 Gbps network: Daejeon (Korea) → Seattle (U.S.A) → Calgary (Canada) → Seattle and Chicago (U.S.A) → Amsterdam (Netherlands) → Stockholm (Five northern European countries) → Moscow (Russia) → Beijing and Hong Kong (China) → Daejeon (Korea). Korea is responsible for the 10-Gbps Pacific Ocean segments that connect Hong Kong and Daejeon and Daejeon and Seattle [4]. KISTI is directly peered to CERN (European Organization for Nuclear Research) via a 10-Gbps network.
Grid in Korea

2.1. Overview

A grid comprises technologies and infrastructure that support the sharing and coordinated use of diverse resources in dynamic, distributed VOs (Virtual Organizations) [5]. There are three types of grids: computing grids, data grids, and access grids (AG). In computing grids, multiple computers from several owners are utilized to run a single very large application. Examples of applications for a computing grid are fluid dynamics and weather. In data grids, multiple storage systems from several owners are used to divide data across the combined resources in order to host one very large data set. An example of an application is HEP (high-energy physics). AG is a collaboration grid that connects multiple collaborative systems belonging to several owners to allow collaboration on a common topic.

We describe the K*Grid project, which was developed using computing and access grids. We also explain grid applications, KOCED (Korea Construction Engineering Development), and Korea@Home.

2.2. K*Grid Project

The K*Grid project was an initiative in grid research and was supported by MIC (Ministry of Information and Communication), Korea. The main goal of the project was to provide an extremely powerful research environment to both universities and industries. The infrastructure is essential for the operation of grid middleware and realizing grid applications. It provided a test-bed and demonstrated the feasibility of grid technology. The objective was to build a large-scale computing grid infrastructure. The project also involved the enhancement of AG and a grid network operation center. The project enabled us to use a huge amount of computing power, virtual experiment facilities, and international collaborations. The project included the development of
national grid infrastructure and grid middleware. The grid infrastructure was connected to supercomputers and high-performance clusters. The total budget of the project for five years (2002–2006) was 37.5 million USD and more than 50 organizations participated in the project. There were two phases in the project.

In the first phase (2002–2004), grid test-beds were constructed and basic grid services were provided from computing centers and university laboratories. As interest in grids has increased, the demand for the production of grid infrastructure has been rapidly growing, not only for the K*Grid project but also for related projects in Korea, such as KOCED and the e-Science project.

The second phase (2005–2006) was carried out to satisfy such requirements. Its objective was to construct a reliable grid service infrastructure that could provide TFlops-level performance. At present, there is a federation of three supercomputing centers: SNU (Seoul National University), PNU (Pusan National University), and KISTI. We have made crucial efforts towards sharing high-performance computing resources.

The K*Grid portal is a web-based grid service platform for providing users with an easy-to-use environment. It was developed based on Java and web service technology. We expect it to play an important role in R&D, as an example of the use of cyberinfrastructure in Korea [2].

The project also included research on the grid itself. The role of grid applications is to migrate to the grid technology frame. The primary goal of the project was to develop grid-based applications to improve the quality of service through the application of grid technology and list the requirements of various applications for further research. Each application project involved testing of the state-of-the-art grid technology, evaluating the performance of the infrastructure, and determining new requirements for further development. Then, by utilizing the advanced grid infrastructure, we performed innovative scientific research and developed high-quality IT services. The K*Grid project includes many grid applications: CFD (computational fluid dynamics), nanomaterial computing, biotechnology, telematics, online games, and rendering. Scientific application had been the major focus until 2004. Subsequently, the focus shifted to IT applications such as online games, telematics navigation services, and rendering services [2].

The first IT application in the K*Grid project was a grid-based telematics navigation service. We developed a high-quality telematics navigation service based on grid technology. It presents realistic 3D image content in a grid environment. It is composed of four elements: a grid-based load balancer, navigation client, 3D image provider, and load monitoring agent.

The second application was a grid-based online game service. The goal was to develop a high-quality online game service in a grid environment and allocate real-time game servers according to the number of concurrent game users. We developed an online game server using a grid pool constructed by grid middleware in order to manage game daemons. It is composed of four elements: a lobby host, room host, data center, and monitoring tool.

The third application was a grid-based rendering service. The goal was to implement either a commercial or non-commercial rendering project in a grid
environment. Locally distributed render farms (Daejeon and Seoul) are interoperated by RenMan™, which manages and realizes a rendering service through a grid.

Since the principle behind a grid is collaboration, the Korean grid research community actively participates in global collaboration. Since global collaboration is also one of the major objectives of grid research, KISTI is a member of OGF (Open Grid Forum), PRAGMA (Pacific Rim Applications and Grid Middleware Assembly), ApGrid (Asia Pacific Grid), and EGEE (Enabling Grid for E-Science). KISTI is also a supporter of related international and domestic collaborations. Many universities and industries in Korea participate in OGF-KR to share computing resources and research information.

2.3. Access Grid

The goal of AG is to connect multiple collaborative systems from several owners to enable collaboration on a common problem. AG activities in Korea started in 2002 as part of the K*Grid project. AG infrastructure in Korea was introduced as a one-room-type node at KISTI. Currently, we have developed over fifty PIG (personal interface to the access grid) and six-room-type nodes that are actively used in areas such as e-Science, bioinformatics, medicine, and meteorology [2].

Currently, researchers in various fields are using AG services. Researchers may use venue and bridge services that are open to the public. Technical support is always available to researchers. In order to improve and provide a higher quality of service, we built a test-bed for an ultra-high definition networked tiled display between KISTI and GIST using a 5-Gbps network. GIST also performs the novel design of an economical videoconference model. To support collaboration among geographically distributed researchers, we provide collaboration tools using AG, EVO, Polycom, and ultra-high definition visualization devices.

2.4. Korea@Home

The next grid application in Korea is Korea@Home. In Korea, Internet-based distributed computing has emerged as a new paradigm of computing style since the late 1990s. There are already thousands of computers that may be available for use in Korea. The goal of this Internet-based distributed computing project, known as “@Home,” is to convince people to allow their computing power that is being currently wasted to be borrowed. The aim of this project is to harness the massive computing power distributed over the Internet [6].

Korea has very advanced high-speed Internet infrastructure with connection speeds of 100 Mbps. This availability of the advanced infrastructure is the advantage of the @Home project. By using this advanced Internet infrastructure, we will build a distributed computing environment for use by the public. This project is known as Korea@Home and is led by the KISTI supercomputing center. The motivation for Korea@Home is that Korea has a higher IT utilization than any other country.

The Korea@Home project started in 2002 and has two sub-goals: (1) to construct a complete Internet-based distributed computing environment and (2) to find applications by using the environment. As an example from 2002, an industry-university joint consortium, including KISTI, carried out a project entitled “The Discovery of the Lead
Compound Generation” in the field of bioinformatics. In the project, an Internet-based distributed computing environment was used to find new drug candidates and calculate the strength of the bonds between known disease proteins and target ligands. We plan to extend this project to various areas. As of November 10, 2008, there were more than 42,000 volunteers. Korea@Home helps achieve a performance of 390 GFlops at maximum capacity and an average of 64 GFlops for a 24-h period [6].

2.5. KOCED

The next grid application is the construction engineering project, KOCED; it is led by MLTM (Ministry of Land, Transport and Maritime affairs) of Korea. The goal of this program is to promote research and development and set up a nationwide education program in order to produce highly qualified researchers and practitioners in the various fields of construction engineering. The ultimate goal of the project is to enhance construction technologies in Korea [7]. The cyberinfrastructure in the project consists of applications, collaboration services, networking, and testing facilities. The goal is to build a dozen or more large-scale testing facilities at major institutes around the country for education and research in construction engineering. Another goal is to connect the testing facilities with a high-performance information network and construct a system that can be controlled and accessed remotely.

The KOCED grid system is a high-speed network service that supports collaboration by linking high-performance computers, large databases, cutting-edge equipment, and other resources located across the country. Figure 2 shows the scheme of the KOCED grid. Through the use of information communication technology, the KOCED grid meets the needs of scattered facilities, namely, to have access to high-speed computing, large-capacity data processing, and advanced equipment for efficient research. The grid reduces the cost for each institute dramatically [7].

![Figure 2. Scheme for the KOCED Grid.](image-url)
KOCED grid portals enable remote users to conduct experiments at KOCED testing facilities. Configured in a web service format, the grid portal allows access to test results and enables remote monitoring of experiments in progress. Cameras installed in the testing facilities also make it possible to observe and verify experiments in real time.

For KOCED grid services, resource management supports the monitoring and location of resources needed by remote users. Long distance researchers can assign their work and obtain results in a consistent manner. Data management enables efficient search and management of experimental data. Telepresence services allow remote access to experimental data. Visual and audio data are transmitted in real time to remote users. If needed, experimental equipment can be operated remotely. Collaboration services enable distributed teams to work together [7].

3. e-Science in Korea

3.1. Definition of e-Science

Thousands of years ago, science was experimental in nature and a way to describe natural phenomena. For the last few hundred years, science is being approached theoretically, as evident in concepts such as Newton’s laws and Maxwell’s equations. For the last few decades, science has become computational in nature, where computations are carried out to simulate complex phenomena. The latest development is e-Science, which is a data-centric science for unifying theory, experiments, and simulation. Currently, science faces many challenging problems that require large resources; in particular, knowledge from many disciplines is required [2]. e-Science is a new research paradigm for computationally intensive scientific research and is carried out in highly distributed network environments [8]. e-Science involves the use of immense data sets that require grid technology [8]. The goal of e-Science is to enable researchers to conduct any study at any time and location.

3.2. e-Science Project in Korea

The main focus of the K*Grid project is the construction of next-generation Internet and business applications. Very little emphasis is laid on scientific applications. In 2005, in order to improve the situation, a Korean e-Science project was started by MEST of Korea, to provide researchers around the country with advanced collaborative environments [2].

The national e-Science project in Korea is aimed at providing innovatively enhanced research infrastructure that enables researchers to use nationally distributed R&D resources such as high-performance computing, databases, scientific instruments, and human resources.

The purpose is to improve the R&D environment. The government promotes projects that help adapt to rapid changes in paradigms for science and technology by enabling global collaboration through e-Science. Therefore, we participated in the standardization of the global e-Science environment. The e-Science infrastructure helps improve R&D activities in various fields in Korea. It is effective for sharing R&D resources nationwide through e-Science.
We have chosen four prior e-Science applications from among many e-science activities. This project also contributed to common software development for e-Science.

3.3. Applications of e-Science

Applications areas of e-Science include physics, engineering, life science, and tele-science. High-energy physics is one application area of e-Science in physics. In the next section, we describe e-Science for high-energy physics in detail.

As an application of e-Science in engineering, we are working on e-AIRS (aerodynamic integrated research systems). The main goal is to establish a powerful and user-friendly collaboration environment for aerospace researchers. Through the user-friendly e-Science grid portal system, remote CFD calculation and experiment management systems are made available. It consists of a CFD framework, remote wind tunnel experiment management system, portal service system, and collaboration system on the AG. The main participants are KISTI, SNU, SMWU (Sookmyung Women’s University), NCSA (National Center for Supercomputing Applications), University of Manchester, and the BAE system. Currently, it is used as both a cyber-education system and remote simulation system. Since 2008, KISTI and WST (Wing Ship Technology) have collaborated on simulations for WIG (wing in ground) craft. The integrated system results in greater research achievements and more effective collaborations. Additionally, it helps reduce costs and R&D time for web portals.

For e-Science applications in life science, we are working on the WISDOM (wide in silico docking on malaria) project. For this project, Chonnam National University participated in the first data challenges on malaria and avian flu in 2007. We worked on AMGA (ARDA metadata grid applications) as a part of WISDOM. AMGA is a metadata catalogue that is included in gLite middleware; this allows access to relational databases with metadata on files or keeps track of tasks on the grid. AMGA is widely used in EGEE and other grid projects. To improve AMGA throughput, we studied multiple AMGA servers and DB connection pool techniques. In 2009, the “nationwide biology data center,” which includes a data farm and the concept of data, will start operations. This project will implemented by adopting grid technology.

Another application of life science is medical research. KISTI supports collaborative tools for research, education, and cures with interactive collaborative facilities for medical research.

Since the goal of e-Science is to enable researchers to study science at any location, tele-science is a good example of an e-Science application that enables remote usage and sharing of very expensive facilities. An example is the HVEM (high voltage electron microscope), which is financially supported by the MEST and has been operating at KBSI since October 2003. The project is aimed at observing experimental data remotely by using an HVEM to collect, store, and manage the data. This project also involves analyzing the data through the online HVEM Grid. The system is designed on the basis of the concept of web services as a powerful collaboration tool in order to control instruments remotely, manage images, control 3-D processing, and store data automatically. The HVEM grid is an integrated system of HVEM and computing resources for HVEM users. Currently, KBSI provides researchers in Korea with access to tele-science facilities. In 2008, researchers at Inha University, which is 170 km away from KBSI, successfully used this facility remotely.
4. e-Science for High-energy Physics

4.1. High-energy physics at KISTI

KISTI is leading the implementation of e-Science projects in Korea. One such application is high-energy physics. In this section, we describe e-Science for high-energy physics in detail. Figure 3 shows the high-energy-physics experiments for e-Science and Figure 4 shows the related projects at the KISTI.

Figure 3. High-energy-physics experiments for e-Science.

Figure 4. High-energy-physics experiments and the related project at the KISTI.
The CDF (Collider Detector at Fermilab) experiment is a Fermilab experiment in the U.S.A. ALICE (A Large Ion Collider Experiment) is one of the LHC (large hadron collider) experiments at CERN in Switzerland. Belle is KEK experiment in Japan. The ILC (International Linear Collider) experiment is a future experiment whose location has not yet been decided. STAR (Solenoid Tracker At RHIC) is part of a BNL (Brookhaven National Laboratory) experiment in the U.S.A.

KISTI is working on the following e-Science projects: Korea-CERN Cooperation for LHC (Large Hadron Collider), NSDC (National Scientific Data Center), FKPPL (France-Korea Particle Physics Laboratory), GLORIAD, and PLSI (Partnership & Leadership Supercomputing Infrastructure), which is supported by MEST, Korea.

4.2. Goal of e-Science for High-energy Physics

High-energy physics experiments are generally conducted at major accelerator sites (onsite) where experimentalists perform detector design, signal processing, data acquisition, and data analysis on a large scale [8]. The goal of e-Science is to enable researchers to study science at any time and location. Therefore, the goal of e-Science for high-energy physics is to enable the study of high-energy physics at any time and location, even if researchers are not present at accelerator laboratories (onsite). High-energy physics requires a particularly well-developed e-Science infrastructure because of the need for adequate computing facilities for the analysis of results and data storage [2]. For computing at the required scale of high-energy physics, we need data grid technology.

4.3. e-Science for High-energy Physics in Korea

4.3.1. Overview

We are working on data production, processing, and analysis to perform high-energy physics experiments onsite at an accelerator laboratory. At Fermilab, data production is performed in shifts at the main control room. Data processing is carried out at the Feynman computing center. Data analysis is carried out in conference rooms. As shown in Figure 5 and Table 2 [9], the processes in e-Science for high-energy physics include data production, processing, and analysis, which can be carried out at any time and location, even without being onsite.

Figure 5. Components of e-Science for high-energy physics. The photographs to the left were taken onsite (Fermilab) and those to the right were taken offsite (KISTI).
First, data collection is performed at both the online and offline shifts from any location. The online shift is carried out from a remote control room, and the offline shift is carried out using an SAM (sequential access through meta-data) data handling system.

Second, data processing is carried out using a high-energy-physics data grid. The objective of the high-energy-physics data grid is to construct a system to process high-energy-physics data and support the high-energy-physics community [8]. Figure 6 shows high-energy-physics data grid farms in Korea. We explain data processing for the ALICE [10], CDF, and CMS experiments in Korea.

![High-energy-physics data grid farms in Korea.](image)

Figure 6. High-energy-physics data grid farms in Korea.
Third, data analysis is carried out by collaborations around the world to analyze and publish their results using collaborative environments.

4.3.2. Data Production

Generally, we collect data onsite, where accelerators are located. However, to adhere to the concept of e-Science, we would like to be able to collect data from any location. One method is to use a remote operation center. An example is the remote operation center at Fermilab, U.S.A., for operating the LHC experiment at CERN. Currently, we have constructed a remote CDF operation center at KISTI that enables CDF users in Korea to take CO (consumer operator) shifts at KISTI in Korea, not at Fermilab, U.S.A. [9]. Figure 7 shows the main control room at Fermilab and the remote control room at KISTI.

![Figure 7. Main control room at Fermilab (left) and remote control room at the KISTI (right).](image)

4.3.3. Data Processing

4.3.3.1. ALICE Experiment

In 2007, MEST, Korea, and CERN entered into an MOU (memorandum of understanding) to build and operate an ALICE farm at KISTI. The LCG organization involves a hierarchy of computing centers from CERN, labeled Tiers 1, 2, and 3.

We built and operated the ALICE Tier 2 center using LCG farms (KR-KISTI-GCRT-01). The ALICE Tier 2 center at KISTI consists of 120 kSI2K CPU and 30 TB of storage. Currently, the farm maintains a 96% operating capacity with 8,000 jobs per month [11]. The ALICE Tier 2 center at KISTI has become a part of the federation of global ALICE farms, which includes 13,804 kSI2K CPU and 99.52 PB of disk space across the world [12]. Currently, around 1,000 physicists from 109 institutes and 31 countries use the ALICE farms [13].
4.3.3.2. CDF Experiment

The increasing luminosity of the Tevatron collider causes the computing requirements for data analysis and MC (Monte Carlo) production to become greater than the dedicated CPU resources that will be available [14,15]. In order to meet future demand, CDF is investigating various computing methods—CAF, DCAF (Decentralized CDF Analysis Farm), and grids. A significant fraction of computing resources is expected to be available to CDF during the LHC era, and CDF can benefit from their use [16].

The regional CDF collaboration of Taiwanese, Korean, and Japanese groups built a CDF analysis farm based on grid farms. We call this federation of grid farms the Pacific CDF analysis farm. Figure 8 shows the components of the farm.

The Pacific CDF analysis farm is a distributed computing model on the grid. It is based on the Condor glide-in concept where Condor daemons are submitted to the grid, effectively creating a virtual private batch pool [17]. Thus, submitted jobs and results are integrated and are shared at grid sites. For work nodes, we use both LCG and OSG (open science grid) farms. The head node of the Pacific CDF analysis farm is located at the Academia Sinica in Taiwan. Currently, the farm has become a federation including one LCG farm at KISTI in Korea (KR-KISTI-HEP), one LCG farm at the University of Tsukuba in Japan (JP-TSUKUBA-U-03), and one OSG and two LCG farms in Taiwan (IPAS_OSG, IPAS_CS, Taiwan-LCG2) [18].

Figure 8. Components of the Pacific CDF Analysis Farm.

4.3.3.3. CMS experiment

The CMS experiment is an experiment at CERN in Switzerland. KISTI supports two CMS data centers in Korea with international network (GLORIAD) [19] and domestic network (KREONET). One is the CMS Tier 2 center at KNU. The other is SSCC (Seoul SuperComputing Center) at the University of Seoul.

KISTI supports a 10-Gbps network between CERN to KNU through GLORIAD and KREONET. The CMS experiment was part of “Load Test 2007.” The targets of
Load Test 2007 were to demonstrate that the connection between a Tier 1 center and each Tier 2 center is sustainable at 10 MB/s for 12 h and that the connection from each Tier 2 center to the Tier 1 center is sustainable at 5 MB/s for 12 h [20]. KNU participated in Load Test 2007 and met the requirements [20]. Table 3 shows the goals and achievements of CSA (computing, software, and analysis challenge) in the CMS experiment [21]. KNU has also participated in CSA07 and CSA08. In CSA07, the transfer rates of CERN to Tier 1 centers were shown to be 13–105 MB/s, depending on the Tier 1 centers. KNU demonstrated cumulative transfer volumes of 80 TB for 17 weeks during CSA07 [20] and 250 TB for 17 weeks during CSA08 [21].

<table>
<thead>
<tr>
<th>Service</th>
<th>CSA08</th>
<th>CSA07</th>
<th>Status</th>
<th>CSA06</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier-0 Reconstruction Rate</td>
<td>150Hz–300Hz</td>
<td>100 Hz</td>
<td>Achieved Bursts</td>
<td>50 Hz</td>
<td>Achieved</td>
</tr>
<tr>
<td>Network Transfers between T0-T1</td>
<td>600 MB/s</td>
<td>300 MB/s</td>
<td>Achieved Bursts</td>
<td>150 MB/s</td>
<td>Achieved (6/7continued)</td>
</tr>
<tr>
<td>Network Transfers between T1-T2</td>
<td>50–500 MB/s</td>
<td>20–200 MB/s</td>
<td>Achieved Most Sites</td>
<td>10–100 MB/s</td>
<td></td>
</tr>
<tr>
<td>Network Transfers T1-T1</td>
<td>100 MB/s</td>
<td>50 MB/s</td>
<td>Only Opportunistic</td>
<td>N/A</td>
<td>Not Attempted</td>
</tr>
<tr>
<td>Job Submission to Tier-1s</td>
<td>50k jobs/day</td>
<td>25k jobs/day</td>
<td>Achieved</td>
<td>12k jobs/day</td>
<td>3k jobs/day</td>
</tr>
<tr>
<td>Job Submission to Tier-2s</td>
<td>105k jobs/day</td>
<td>75k jobs/day</td>
<td>20k jobs</td>
<td>48k jobs/day</td>
<td>Achieved</td>
</tr>
<tr>
<td>MC Simulation</td>
<td>$1.5 \times 10^9$ events/year</td>
<td>50M events/ month</td>
<td>Achieved</td>
<td>N/A</td>
<td>Not Attempted</td>
</tr>
</tbody>
</table>

In the meantime, SSCC is part of the University of Seoul and operates the CMS HI (heavy ion) data center. SSCC was established in 2003 with a funding of one million USD [22]. The middleware of the SSCC farm is the OSG (open science grid) farm since the SSCC farm receives CMS HI data from the Tier 1 center at Fermilab in the U.S.A. Most grid sites in the U.S.A. are OSG farms since the OSG project is funded by the U.S. government. KISTI supports a 1-Gbps network between Fermilab, U.S.A., and SSCC through GLORIAD and KREONET.

4.3.3.4. Results of Data Processing

Figure 9 shows the accounting data for LCG farms in Korea. KISTI runs the KR-KISTI-GCRT-01 farm for the ALICE experiment and the KR-KISTI-HEP farm for the CDF experiment. KNU runs the LCG_KNU farm for the CMS experiment.
Figure 9. Accounting data for LCG farms in Korea.

Figure 10 shows the total CPU time used per VO in Korea. The dominant CPU is used for the ALICE experiment.

4.3.4. Data Analysis

For the data analysis collaboration, we hosted the EVO server systems at KISTI so that high-energy physicists in Korea can use it directly without using reflectors in the U.S.A. Therefore, the EVO server at the KISTI enables global collaborations for performing analysis and easily publishing results together [18]. For the data analysis collaboration, we constructed EVO servers at KISTI. When users in Korea use EVO servers at KISTI, the routing time is reduced by 60 ms with no congestion from the network inside the U.S.A, which provides a very stable research environment [18].
5. Global Collaborations

The term e-Science is used to describe computationally intensive science that is carried out in highly distributed network environments or science in which immense data sets that require grid computing are used. The term sometimes includes technologies that enable distributed collaboration, such as access grids. As an example of well-suited applications, physicists have particularly well-developed e-Science infrastructure due to their need for adequate computing facilities for results analysis and data storage. The goal of e-Science in physics is to study physics at any time and from any location. Therefore, we held “The International Workshop on e-Science for Physics 2008” at Daejeon Convention Center, Korea, on September 8–9, 2008, to share the concept of e-Science in the areas of physics and to build a global physics community. Areas of interest include particle physics and nuclear physics, numerical relativity and astrophysics, fluid physics, nano-physics, biophysics, medical physics, and nuclear fusion research. Seventy two papers from five countries were presented. Outstanding papers from the workshop will be published in JKPS (Journal of the Korean Physical Society).

We have collaborated with many international particle physics and nuclear physics communities. One of them is IN2P3 in France. FKPPL (France-Korea Particle Physics Laboratory) was established in March 2008. The objective of FKPPL is to carry out joint cooperative activities, or “joint research projects,” under a scientific research program in the fields of high-energy physics, e-Science, and related technologies recommended by its steering committee.

The projects of the scientific program selected from the joint research proposals are listed in Table 4.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Leading Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Directors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>France (IN2P3)</td>
</tr>
<tr>
<td></td>
<td>Korea (KISTI)</td>
</tr>
<tr>
<td>Grid Computing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>France (IN2P3)</td>
</tr>
<tr>
<td></td>
<td>Korea (KISTI)</td>
</tr>
<tr>
<td>ILC Detector R&amp;D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>France (IN2P3)</td>
</tr>
<tr>
<td></td>
<td>Korea (KISTI)</td>
</tr>
<tr>
<td>ILC Electronics</td>
<td></td>
</tr>
<tr>
<td></td>
<td>France (IN2P3)</td>
</tr>
<tr>
<td></td>
<td>Korea (KISTI)</td>
</tr>
<tr>
<td>CDF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>France (IN2P3-CNRS)</td>
</tr>
<tr>
<td></td>
<td>Korea (KISTI)</td>
</tr>
</tbody>
</table>

We are working on the CDF-LIA project. The goal of this project is to develop collaboration between the French and Korean computing centers in the CDF experiment. The project has two main objectives: to integrate KISTI into the LCG and to foster joint research on CDF experiments as an e-Science application.
The higher instantaneous luminosity of the Tevatron requires a large amount of computing resources for both data analysis and MC production in CDF experiments. Therefore, serious efforts have been made to enable access to shared grid resources over the last two years [23]. LPNHE, along with CCIN2P3, was involved in an effort to extend the dedicated resource-based computing model by using the Condor glide-in mechanism to create dynamic Condor pools on top of existing batch systems, without the need to install any additional software. All the monitoring tools supported on the dedicated farms, including semi-interactive access to the running jobs and detailed monitoring, were preserved. Thus, Grid CAF came into existence. The first production Grid CAF, developed and deployed at the Tier1 center at CNAF, Bologna, was followed by the deployment of another production Grid CAF at the French Tier 1 center at Lyon [23]. The main aim is to collaborate for the development of this framework at both the French Tier 1 center and KISTI.

The other purpose of this project was to collaborate on two fundamental topics in high-energy physics, namely, $B$ and top physics at the CDF experiment, where both teams are participating. $B$ physics is a major topic in heavy-flavor physics; the French IN2P3 team is focusing its efforts on a charge-parity violation and $B_s$ mixing in particular. We still expect important breakthroughs in the coming years. Another topic of interest is to collaborate on top physics, in particular, the top decay into tau leptons, which is instrumental for physics beyond the standard model. For these two experimental searches, large MC productions are essential tools [23].

In conclusion, within the framework of this project, KISTI and LPNHE-IN2P3 conducted joint research on CDF as an e-Science application. Research focused on CDF grid technology, and $B$ and top physics [23].

6. Conclusions

For developing cyberinfrastructure in Korea, we started the grid project in 2002 and the e-Science project in 2005; in these projects, the computing and network resources at KISTI were used. The K*Grid project demonstrated the use of computing and access grids. The applied projects of the grid are Korea@Home and KOCEDE. Then, we arrived at the stage of e-Science. Through KISTI, e-Science in Korea has become a global science gateway. The application areas of e-Science are physics, engineering, life science, and tele-science.

High-energy physics is a well-suited application of e-Science projects in Korea. We succeeded in developing and installing an e-Science environment for high-energy physics, and we lead the community. The processes in e-Science for high-energy physics are data production, processing, and analysis. For data production, we operate a remote control room. For data processing, KISTI operates an ALICE Tier 2 center and Pacific CAF, while KNU operates a CMS Tier 2 center. KISTI also participated in a CDF experiment and installed a CDF grid farm in collaboration with a federation from Taiwan and Japan. For data analysis, we provided the high-energy physics community with an EVO server. We also participated in global collaborations. In conclusion, high-energy physics demonstrates the great success of e-Science projects in Korea.
Acknowledgments

We would like to thank Minho Jeung and Hyunwoo Kim (KISTI) for data production and processing, Soonwook Hwang (KISTI) for EGEE, Yuchul Yang (KNU), Mark Neubauer (UCSD), and Frank Wüerthwein (UCSD) for Central Analysis Farm, and Igor Sfillioi (Fermilab) and Hsieh Tsan Lung (Academia Sinica) for CDF Grid Farm.

References

[1] See Wikipedia home page: 
## Subject Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm design</td>
<td>27</td>
</tr>
<tr>
<td>Amazon Cloud</td>
<td>192</td>
</tr>
<tr>
<td>applications</td>
<td>141</td>
</tr>
<tr>
<td>asynchronous distributed computing</td>
<td>74</td>
</tr>
<tr>
<td>CCR</td>
<td>311</td>
</tr>
<tr>
<td>cerebro-vascular blood flow</td>
<td>365</td>
</tr>
<tr>
<td>Chemomentum</td>
<td>157</td>
</tr>
<tr>
<td>Cholesky</td>
<td>3</td>
</tr>
<tr>
<td>cloud computing</td>
<td>177, 192, 216, 236, 251, 267</td>
</tr>
<tr>
<td>cluster and grid computing</td>
<td>177</td>
</tr>
<tr>
<td>clustering</td>
<td>311</td>
</tr>
<tr>
<td>collaborative applications</td>
<td>299</td>
</tr>
<tr>
<td>coordination languages</td>
<td>74</td>
</tr>
<tr>
<td>cost effective</td>
<td>236</td>
</tr>
<tr>
<td>curation</td>
<td>399</td>
</tr>
<tr>
<td>data lifecycle</td>
<td>399</td>
</tr>
<tr>
<td>data locality</td>
<td>55</td>
</tr>
<tr>
<td>data management in workflows</td>
<td>192</td>
</tr>
<tr>
<td>data processing</td>
<td>464</td>
</tr>
<tr>
<td>data replication</td>
<td>399</td>
</tr>
<tr>
<td>DEISA</td>
<td>141, 157</td>
</tr>
<tr>
<td>D-Grid</td>
<td>157</td>
</tr>
<tr>
<td>distributed file I/O</td>
<td>399</td>
</tr>
<tr>
<td>Distributed Hash Tables</td>
<td>109</td>
</tr>
<tr>
<td>distributed systems</td>
<td>216</td>
</tr>
<tr>
<td>D-MON</td>
<td>157</td>
</tr>
<tr>
<td>Dynamic Querying</td>
<td>109</td>
</tr>
<tr>
<td>e-infrastructure</td>
<td>157, 451</td>
</tr>
<tr>
<td>Energy Sciences Network (ESnet)</td>
<td>422</td>
</tr>
<tr>
<td>enterprise frameworks for</td>
<td></td>
</tr>
<tr>
<td>cloud computing</td>
<td>267</td>
</tr>
<tr>
<td>e-Science</td>
<td>299, 451, 464</td>
</tr>
<tr>
<td>exascale</td>
<td>95</td>
</tr>
<tr>
<td>financial application</td>
<td>380</td>
</tr>
<tr>
<td>FPGA</td>
<td>55</td>
</tr>
<tr>
<td>grid</td>
<td>109, 157, 464</td>
</tr>
<tr>
<td>grid computing</td>
<td>141, 251, 380, 451</td>
</tr>
<tr>
<td>high energy physics</td>
<td>464</td>
</tr>
<tr>
<td>high performance computing</td>
<td>55</td>
</tr>
<tr>
<td>high performance computing</td>
<td>95, 141, 216, 451</td>
</tr>
<tr>
<td>interconnection networks</td>
<td>95</td>
</tr>
<tr>
<td>license management</td>
<td>216</td>
</tr>
<tr>
<td>linear algebra</td>
<td>3</td>
</tr>
<tr>
<td>loop transformations</td>
<td>55</td>
</tr>
<tr>
<td>MapReduce</td>
<td>311</td>
</tr>
<tr>
<td>matrix factorization</td>
<td>3</td>
</tr>
<tr>
<td>middleware</td>
<td>157</td>
</tr>
<tr>
<td>MPI</td>
<td>311</td>
</tr>
<tr>
<td>multicore</td>
<td>3</td>
</tr>
<tr>
<td>multidimensional scaling</td>
<td>311</td>
</tr>
<tr>
<td>network planning</td>
<td>422</td>
</tr>
<tr>
<td>networks for large-scale science</td>
<td>422</td>
</tr>
<tr>
<td>OGSA</td>
<td>157</td>
</tr>
<tr>
<td>OMII-Europe</td>
<td>157</td>
</tr>
<tr>
<td>on-demand resource provisioning</td>
<td>177</td>
</tr>
<tr>
<td>PaaS</td>
<td>216</td>
</tr>
<tr>
<td>Peer-to-Peer</td>
<td>109</td>
</tr>
<tr>
<td>performance</td>
<td>311</td>
</tr>
<tr>
<td>personal clusters</td>
<td>192</td>
</tr>
<tr>
<td>portfolio optimization</td>
<td>380</td>
</tr>
<tr>
<td>QR</td>
<td>3</td>
</tr>
<tr>
<td>scheduling</td>
<td>3</td>
</tr>
<tr>
<td>scheduling techniques</td>
<td>27</td>
</tr>
<tr>
<td>scientific workflows</td>
<td>192</td>
</tr>
<tr>
<td>service oriented computing</td>
<td>267</td>
</tr>
<tr>
<td>software engineering</td>
<td>267</td>
</tr>
<tr>
<td>stream processing</td>
<td>74</td>
</tr>
<tr>
<td>supercomputing</td>
<td>157</td>
</tr>
<tr>
<td>system level acceleration</td>
<td>365</td>
</tr>
<tr>
<td>system-level science</td>
<td>299</td>
</tr>
<tr>
<td>task graph</td>
<td>3</td>
</tr>
<tr>
<td>UNICORE</td>
<td>157</td>
</tr>
<tr>
<td>utility computing</td>
<td>251</td>
</tr>
<tr>
<td>VCL</td>
<td>236</td>
</tr>
<tr>
<td>ViroLab</td>
<td>299</td>
</tr>
<tr>
<td>virtual data</td>
<td>399</td>
</tr>
<tr>
<td>virtual laboratory</td>
<td>299</td>
</tr>
<tr>
<td>workflows</td>
<td>399</td>
</tr>
<tr>
<td>WS-RF</td>
<td>157</td>
</tr>
</tbody>
</table>
Author Index

Abramson, D. 399
Averitt, S. 236
Badia, R. 3
Bae, S.-H. 311
Benoit, A. 27
Beraldi, P. 380
Bergmann, S. 157
Berriman, B. 192
Beyls, K. 55
Breu, R. 157
Bubak, M. 299
Buuya, R. 267
Cappello, F. 127
Catalyurek, U.V. 341
Chaniotakis, E. 422
Cho, K. 464
Chu, X. 267
D’Hollander, E.H. 55
Daivandy, J. 157
Dalheimer, M. 216
Dart, E. 422
David, T. 365
Deelman, E. 192
Demuth, B. 157
Dongarra, J. 3
Dreher, P. 236
Ekanayake, J. 311
Epicoco, I. 380
Fox, G. 311
Gentzsch, W. v, 141
Giesler, A. 157
Grandinetti, L. v, 380
Gubala, T. 299
Guok, C. 422
Guercan, M. 341
Hagemeier, B. 157
Hartley, T.D.R. 341
Herault, T. 127
Ho, T. 399
Holl, S. 157
Huber, V. 157
Jan, M. 127
Johnston, W. 422

Joubert, G. v
Juve, G. 192
Kasztelnik, M. 299
Kee, Y.-S. 192
Kurzak, J. 3
Largillier, T. 127
Lindner, M. 251
Lippert, T. 157
Livny, M. 192
Llorente, I.M. 177
Ltaief, H. 3
Malawski, M. 299
Mallmann, D. 157
Marchal, L. 27
Massonet, P. 251
Memon, A.S. 157
Memon, M.S. 157
Menday, R. 157
Metsch, T. 251
Metzger, J. 422
Montero, R.S. 177
Moreno-Vozmediano, R. 177
Öster, P. 451
Peyronnet, S. 127
Pfreundt, F.-J. 216
Qiu, X. 311
Quetier, B. 127
Rambadt, M. 157
Riedel, M. 157
Romberg, M. 157
Roth, P.C. 95
Ruiz, A. 341
Saltz, J. 341
Schuller, B. 157
Sertel, O. 341
Shafarenko, A. 74
Sills, E. 236
Singh, G. 192
Streit, A. 157
Talia, D. 109
Tierney, B. 422
<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tipparaju, V.</td>
<td>95</td>
<td>Violi, A.</td>
<td>380</td>
</tr>
<tr>
<td>Trufio, P.</td>
<td>109</td>
<td>Vivien, F.</td>
<td>27</td>
</tr>
<tr>
<td>Ujaldon, M.</td>
<td>341</td>
<td>Vouk, M.A.</td>
<td>236</td>
</tr>
<tr>
<td>Vaquero, L.M.</td>
<td>251</td>
<td>Yu, W.</td>
<td>95</td>
</tr>
<tr>
<td>Vecchiola, C.</td>
<td>267</td>
<td>Yuan, H.</td>
<td>311</td>
</tr>
<tr>
<td>Vetter, J.S.</td>
<td>95</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>