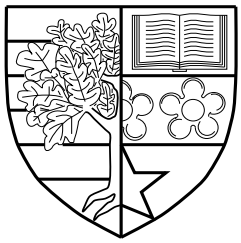
Implementing High-Level Parallelism on Computational Grids

by

Abdallah Deeb I. Al Zain



Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
School of Mathematical and Computer Sciences
June 2006

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Abdallah Deeb I. Al Zain (Candidate)

Phil Trinder, Greg Michaelson (Supervisors)

Date

In the name of GOD, Most Gracious, Most Merciful,

≪It is GOD Who brought you forth from the wombs of your mothers when ye know nothing; and He gave you hearing and sight and intelligence and affection: that ye may give thanks (to GOD)≫ (Qur'an, 16:78)

Abstract

Special purpose high performance computers are expensive and rare, but workstation clusters are cheap and becoming common. Emerging technology offers the opportunity to integrate clusters into a single high performance computer - a computational GRID. The acceptance of computational GRIDs, however, is seriously hampered by the difficulty of efficiently managing the parallelism in such heterogeneous clusters, with characteristics radically different from a conventional high performance computer. To program this complex and dynamic architecture effectively we propose to use a language with high-level constructs, GPH, and to extend its runtime environment, GUM.

The first contribution of this thesis is to develop *GRID-GUM*1, an initial port of GUM to computational GRIDs. Systematic evaluation shows that *GRID-GUM*1 delivers acceptable speedups on relatively low latency and on homogeneous computational GRIDs. However for high latency or heterogeneous computational GRIDs poor load scheduling limits performance.

We next present an adaptive runtime environment *GRID-GUM*2, which includes monitoring mechanisms that determines static and dynamic properties of the underlying clusters and an adaptive scheduling mechanism that dynamically modifies parallel execution accordingly. To the best of our knowledge, *GRID-GUM*2 is one of the first fully implemented virtual shared memory runtime environment on the GRID. Evaluating *GRID-GUM*2's performance demonstrates that virtual shared memory is feasible on computational GRIDs and that it can deliver good speedups if combined with an aggressive dynamic load distribution mechanism.

Acknowledgement

All praises are due to God for His boundless bounties and, in particular, for giving me the good health and the strength of determination to enable me to carry out this work.

I record my indebtedness to my great friend, mentor and supervisor, Greg Michaelson. A second father to me through his friendship and support this thesis was accomplished.

One of the sweetest fruits of life to me is Friendship, and for my last three years in my PhD I have been blessed with a few true friends: The hospitable, independent and warm loving brother to me Gonzalo Muyo and his lovely family (Ana & Elena), an Epicurean by nature, whose purity of soul is unparalleled. The mellifluous philosopher, with whom I disagreed sometimes but never argued. The, whose enthusiasm for life sparked a wild laughter that proved the best medicine for my heart in turbulent times; always by my side in happiness and sadness. My dear Ioannis, a man endued with integrity and rare powers of endurance; our problems always weighed less in our hearts once shared with each other. I can not forget the warmth and hospitality you shown to me with no calculation, and demonstrate how it is a greater pleasure to give than to receive

My Five years at Heriot-Watt introduced me to many friends who deserve a special mentioned for having created the right atmosphere for me to keep going on my research: Manuel Silva, Manuel Maarek, Simona Gagliardi, Andre De Bu Bois, Gudmund Grov, and Margaret Robinson (my Scottish mum). To the best time of my life I spent at 15 Mentone Terrace with the best flatmates Gonzalo and Andre.

And here in Edinburgh, I found the ancient Greek's gods have born like the ancient goddess of Love, (Venus). I found this god is still alive today, like this beautiful nymphs who I met and bewitched me: with her beautiful face, Eva, few things in life are as worth as her beautiful eyes whom I so much loved! She is a very special girl for whom my only regret is that I have not met her longtime ago. It is impossible to describe in a few words my gratitude to her for the happiness

she gave me, and the simple but deep joys of sharing in Love.

It is now the time to focus on people and colleagues who were more directly related with my PhD. I would like to thank my supervisor, Phil Trinder, for his guidance and time-efficiency in dealing with my thesis and also for the financial support which covered my tuition fees during my PhD. I am also indebted to Phil Trinder for the fruitful weekly supervision meeting which helped me to finish this thesis and made work motivation. I am very thankful to my supervisor Hans-Wolfgang Loidl who has been an invaluable person to discuss the implementation of GUM and many technical issues. Also I am much indebted again to my supervisor Greg Michaelson for providing me with hands-on advice research-wise; always supporting and encouraging. My office mates Ioannis, Andre, Xiao Yan, and Zara have also made work pleasant at lab G.59. I am very grateful to the people in the department of computing at Heriot-Watt University for providing valuable assistance. I am very thankful to my examiners David Duke and Peter King for their time and their useful comments. I am also grateful to many others, all of whom can not be named.

And last but not least, to my parents who I register my profound gratitude to them who have trained me, taught me, amongst other things, to take life easy while encourage me to learn to face, on my own, the multi-faceted challenge of life. To my brothers, my sister and their families for all their emotional support, understanding and above all their limitless love. To my country which I have not seen (Palestine) and to my people who are suffering unlawful occupation. This thesis is therefore dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Ethos

Special High Performance Computers are expensive and rare resources, but workstation clusters are cheap and becoming common. Emerging GRID technology offers the opportunity to integrate appropriately enabled high performance computers into a single high performance computer, and pervasive clusters are an obvious target. However, such architectures raise several technical challenges including the following. A network of high performance computers is much harder to utilise effectively than a typical dedicated, flat and homogeneous high performance computer. Communication latency is much higher, and it may be comprised of heterogeneous components: typically a network of clusters of varying sizes and performance. Moreover the interconnect is shared and hence the effective speed the architecture may vary unpredictably.

1.2 Contributions

The thesis proposes a novel framework to run a single large program on computational GRIDs, with the associated challenges of partitioning the program and managing communication and synchronisation between concurrent tasks. In particular we propose in this thesis to *use a language with high-level parallel coordination (*GPH*) that abstracts over the complexities of the architecture, and*

develop a runtime environment that automatically adapts the GRID *architecture (GRID-GUM2).*

The thesis makes the following contributions.

1. The GUM runtime environment is modified for deployment over a wide area network, realising *GRID-GUM*1 [AZTML03]. In addition, we conduct a systematic series of *GRID-GUM*1 performance measurements on different configurations of computational GRIDs with widely varying latencies, and programs with different communication degree, comparing the performance under PVM and MPICH communication libraries with the GRID implementation of the MPI standard MPICH-G2 [AZTLM04].

2. To overcome the limitations of *GRID-GUM*1, *GRID-GUM*2 has been designed with novel dynamic load scheduling mechanisms for shared hierarchical heterogeneous computational GRIDs. *GRID-GUM*2 is the first fully implemented virtual shared memory runtime environment that dynamically manages parallel execution on computational GRIDs. (Virtual shared memory is a programming model, allows processors on a distributed-memory machine to be programmed as if they had shared memory, Section 2.4.5) [AZTLM05, AZMLT04].

3. This thesis demonstrates that lightweight adaptive load distribution techniques, like those in *GRID-GUM*2, can deliver good performance for a typical set of applications on both high- and low- latency, and both homogeneous and heterogeneous computational GRIDs [AZTLM06].

1.3 Thesis Structure

The structure of this thesis is as follows.

Chapter 2 describes related work including GRID technology, load distribution, functional language, and GUM, a parallel functional language runtime environment for GPH.

Chapter 3 details the design and implementation of adapting the GUM parallel runtime environment for execution on the Globus Toolkit GRID middle-ware, *GRID-GUM*1. It includes an evaluation of the performance of *GRID-GUM*1 and presents some of the first systematic performance measurements of several high-level parallel programs on both homogeneous and heterogeneous computational GRIDs.

Chapter 4 presents the design and the implementation of the new adaptive runtime environment *GRID-GUM*2. In particular this chapter describes details of *GRID-GUM*2's monitoring mechanism and the adaptive load distribution mechanism.

Chapter 5 evaluates *GRID-GUM*2 on a range of computational GRID architectures and with programs with varying characteristics. It also demonstrates that virtual shared memory (Section 2.4.5) is feasible on GRID architectures and that it can deliver good speedups if combined with an aggressive dynamic load distribution mechanism. It also presents measurements that quantify the improvements on a small, but realistic, GRID architecture. In particular, it evaluates the performance of the new adaptive load distribution mechanism of *GRID-GUM*2 on a range of computational GRID configurations. On relatively low latency networks and homogeneous and heterogeneous architectures. Furthermore, on high latency networks and homogeneous and heterogeneous architectures. Finally, it measures *GRID-GUM*2's scalability on a high latency heterogeneous architecture.

Chapter 6 summarises the original contributions to knowledge that this thesis has made. Further research directions for improving effectiveness and efficiency of high level parallelism in computational GRIDs are also identified.

1.4 Authorship

Unless otherwise stated, the work reported throughout this thesis including the following research publications were done by the author with the contributions of his supervisors Dr. P. W. Trinder, Dr. G. J. Michaelson and Dr. H-W. Loidl.

1. A. Al Zain, P. Trinder, H-W. Loidl and G. Michaelson. Managing Heterogeneity in a GRID Parallel Haskell. In *Journal of Scalable Computing: Practice and Experience*, Vol 6, No 4, 2006.

2. A. Al Zain, P. Trinder, H-W. Loidl and G. Michaelson. Managing Heterogeneity in a GRID Parallel Haskell. In V. Sunderam, D. van Albada, P. Sloot, J. Dongarra, editors, *International Conference on Computer Science (ICCS05)*, LNCS. Springer, 2005.

3. A. Al Zain, G. Michaelson, H-W. Loidl and P. Trinder. Improving Load Balance in a GRID-Enabled Parallel Haskell. In *TFP'04 – International Workshop on Trends in Functional Programming*, Draft Proceedings, München, Germany, pp329–344, November 2004.

4. A. Al Zain, P. Trinder, H-W. Loidl and G. Michaelson. *GRID-GUM*: Towards GRID-enabled Haskell. In *IFL'04 – International Workshop on the Implementation of Functional Languages*, Draft Proceedings, Lübeck, Germany, pp221–238, September 2004.

5. A. Al Zain, P. Trinder, G. Michaelson and H-W. Loidl. *GRID-GUM*: Haskell on GRIDs. In *TFP'03 – International Workshop on Trends in Functional Programming*, Draft Proceedings, Edinburgh, UK, pp223–238, September 2003.

Chapter 2

Background

2.1 Computational GRIDs

2.1.1 Introduction

The GRID is the computing and data management infrastructure that is intended to provide the electronic underpinning for a global society in business, government, research, science and entertainment [FK99b, For05, Glo05, BFH03a]. The idea behind the GRID is to serve as an enabling technology for a broad set of applications in different areas.

Driven by increasingly complex problems and propelled by increasingly powerful technology, today's science is as much based in computation, data analysis, and collaboration as on the efforts of individual experimentalists and theorists. But even as computer power, data storage and communication continue to grow exponentially, computational resources are failing to keep up with what scientists demand of them. Foster argues that personal computer in 2002 is as fast as a supercomputer of 1990, but 10 years ago, biologists were happy to compute a single molecular structure. Now, they want to calculate the structures of complex assemblies of macromolecules and screen thousands of drugs candidates. Personal computers now ship with up to 100 gigabytes of storage - as much as an entire 1990 supercomputer centre. By 2006, several physics projects will produce multiple petabytes of data per year [Fos02a]. Some wide area networks now operate at

155 megabits per second, three orders of magnitude faster than the state-of-the-art 56 kilobits per seconds that connected U.S. supercomputer centres in 1985. But to work with colleagues across the world on petabyte data sets, scientists now demand tens of gigabits per second.

What many term the 'computational GRID' offers a potential means of surmounting these obstacles to progress. Built on the Internet and the World Wide Web, the computational GRID is a new class of infrastructure. By providing scalable, secure, high-performance mechanisms for discovering and negotiating access to remote resources, the GRID promises to make it possible for scientific collaborations to share resources on an unprecedented scale and for geographically distributed groups to work together in ways that previously impossible [FKT01, Fos02b].

2.1.2 Evolution

Computational GRIDs have evolved in three stages: first-generation systems that were the forerunners of GRID computing as it recognised these days; second-generation systems with a focus on middle-ware to support large-scale data and computation; and third-generation systems in which the emphasis shifts to distributed global collaboration, a service-oriented approach and information layer issues [DRBJ02].

The next three subsections discuss these three stages:

The First Generation (Early 1990s)

The first generation efforts started as projects to link supercomputing sites and this approach was known as metacomputing. The typical objective of the metacomputing projects was to provide computational resources to a range of high performance applications. Two representative projects in the vanguard of this type of technology were FAFNER [FAF05] and I-WAY [FGN+96].

The Second Generation (Late 1990s)

The emphasis of the first generation efforts in GRID computing was in part driven by the need to link supercomputing centre. Metacomputing, especially the I-WAY project [DFP+96], successfully achieved this goal. However, today the GRID infrastructure is capable of binding together more than just a few specialised supercomputing centres. The improvement in the network technology allowed the GRID to be viewed as a viable distributed infrastructure on a global scale that can support diverse applications requiring large scale computation and data. This vision of the GRID was presented as middle-ware computing [FK99b]. Middle-ware is generally considered to be the layer of software sandwiched between the operating system and applications. Recently, middle-ware computing reemerged as a means of integrating software applications running in distributed heterogeneous environment.

The most significant to date projects which present second generation can be classified under:

- GRID core technology projects like Globus Toolkit [FK97], described in Section 2.1.4, and *Legion* [GLFK98].
- Distributed object systems which include projects like *Jini* and *RMI* [Jin04] and *CORBA* [AGG+99].
- GRID resource schedulers and brokers which include projects whose primary focus is batching and resource scheduling like, *Condor* [Con05], the *portable batch system* (PBS) [PBS05], the *Sun Grid Engine* (SGE) [SGE] and the *load sharing facility* (LSF) [Com05, ZZWD93], and projects with general perspective of GRID resource brokers and schedulers like *storage resource broker* (SRB) [RM01] and *Nimrod-G* [BAG00, AGK00].
- GRID portals applications which allow scientists and researchers to access resources specific to a particular domain of interest via a Web interface, also provide access to GRID resources: *NPACI HotPage* [Hot05], *SDSC Grid port toolkit* [SDS05] and *Grid portal development kit* [Kit05].

- Integrated systems which include projects dedicated to a number of exemplar high-performance wide-area applications. The most representative projects are *Cactus* [ADF+01], *DataGrid* [Dat05], *UNICORE* [AS99] and *WebFlow* [AFFH98, HAFF99].

- Peer-to Peer computing as implemented, for example, by *Napster* [Nap05], *Gnutella* [Gnu05], *Freenet* [CSWH01] and project *JXTA* [JXT05].

In the second generation, the core software for the GRID has evolved from that provided by the early vanguard offering, such as Globus Toolkit (GT1) and Legion, which where dedicated to the provision of proprietary services to large and computationally intensive high-performance applications, through to the more generic and open deployment of (GT2). Alongside this core software, the second generation also saw the development of a range of accompanying tools and utilities, which were developed to provide high level services to both users and applications.

The Third Generation (2002)

The second generation provided the interoperability that was essential to achieve large scale computation. As further GRID solutions were explored, other aspects of the engineering of the GRID became apparent. In order to build new GRID applications it was desirable to be able to reuse existing components and information resources and to assemble these components in a flexible manner. The solutions involved increasing adoption of a service-oriented model which includes Web services and agent-based computing and increasing attention to metadata. The two main representative applications in the third generation are:

- *World Wide Web Consortium (W3C)* which aims to establish a web service standards which include: *SOAP (XML protocol)*, *Web Services Description Language (WSDL)* [WSD05] and *Universal Description Discovery and Integration (UDDI)* [UDD05].

- *The Open Grid Services Architecture (OGSA) framework* [FKNT02]. OGSA presents the Globus-IBM vision for the conversions of Web services and GRID computing.

In short, the third generation shifts the focus in describing the GRID as the infrastructure of e-Science rather than the enabling technology of large-scale data and computation.

At the start of this thesis in 2002 most of the third generation applications were still under development and the ones were released were still unstable like Globus Toolkit 3 (GT3). Due to this, this thesis uses the most reliable and fully implemented application in the second generation, Globus Toolkit 2 (GT2) which is fully described in Section 2.1.4.

2.1.3 Computational GRID Applications

This sub-section discuses the applications which use the most mature GRID second generation implementations. According to Foster in [FK99a], one can identify three broad major application classes for computational GRIDs:

- *Distributed Supercomputing* applications use GRIDs to aggregate substantial computational resources in order to tackle a problem that cannot be solved on a single system. For example, the accurate simulation of complex physical processes which can require high spatial and temporal resolution in order to resolve fine-scale detail. Distributed supercomputing has been used sucessfully in cosmology [NBG+96], high-resolution ab initio computational chemistry computations [NH96], and climate modeling [MMF+93].

- *High-Throughput Computing* applications where the GRID is used to schedule large numbers of loosely coupled or in dependent tasks, with the goal of putting unused processor cycles from idle workstations to work. The result may as in distributed supercomputing , the focusing of available resources on a single problem, but the independent nature of tasks involved leads to

very different types of problems and problem-solving methods. The most obvious example of a high-throughput system is the Condor System [BL99].

- *Data-Intensive Computing* applications where the focus is on synchronising new information from data that is maintained in geographically distributed repositories, digital libraries, and databases. This synthesis process is often computationally and communication intensive as well. An example for data-intensive computing application is, high-energy physics experiments which generates terabytes of data per day, or around a petabyte per year [DFF+02].

There are other two application classes which are related to computational GRIDS:

- *On-Demand Computing* applications which use GRID capabilities to meet short-term requirements for resources that can not be cost-effectively or conveniently located locally. These resources may be computation, software, data repositories, specialised sensors, and so on. The NEOS [JC01] and NetSolve [CD95] present examples of on-demand computing application.

- *Collaborative Computing* applications which are concerned primarily with enabling and enhancing human-to-human interactions. Such applications are often structured in terms of virtual shared space. Many collaborative applications are concerned with enabling the shared use of computational resources such as data archives and simulations. For example, the Boiler-Maker system developed at Argonne National Laboratory allows multiple users to collaborate on the design of emission control systems in industrial incinerators [DFH+96].

This thesis targets distributed supercomputing applications. It focuses on solving large computational problems which requires lots of computation power. The idea is to aggregate substantial computational resources in the GRID in order to tackle each single problem. This methodology introduces new challenging issues

including the need for an adaptive load distribution mechanism to schedule resources of many PEs, latencies between these PEs, and achieving and maintaining high levels of performance across heterogeneous systems. Broadly speaking this thesis introduces one of the first fully implemented and tested adaptive load distribution mechanisms over a wide area network using a virtual shared memory to tackle these challenging issues in computational GRID architecture as presented in chapters 4 and 5.

2.1.4 Globus Toolkit

In this section, we provide a detailed description of the Globus Toolkit computational GRID implementation. Globus project is developing an integrated set of basic grid services, termed the Globus Toolkit. The Globus approach differs from the other computational GRID implementations described in Section 2.1.2, in three ways [Fos99]:

- It provides a bag of services model, which allows applications to use GRID services without having to adopt a particular programming model.
- It has provision of specialised mechanism that may coexist with, but sometimes replace, mechanisms provided by high performance computing applications.
- It includes support for an information based approach to meeting application performance requirements.

For these reasons, and the stability, full implementation and the continuous implementation support from the Globus Team [Glo05, AHS+], Globus Toolkit has been chosen as the middle-ware implementation in this thesis.

The Globus Toolkit is open source software with an open architecture [FK97]. It is a collection of software components designed to support the development of applications for high performance distributed computing environments [FK98b]. The three main components are:

- Resource Management: allocation and management of Grid resources;
- Information Services: providing information about Grid resources;
- Data Management: accessing and managing data in a Grid environment.

All of these components use services provided by the Grid Security Infrastructure (GSI) protocol at the connection layer.

GRID Security Infrastructure (GSI)

As the Globus Toolkit can connect different sites over open networks to GRID systems, there is the need for mechanisms that enable authentication and secure communication. The bases for GSI are public key encryption, X.509 certificates and the Secure Socket Layer communication protocol. The GSI implementation of the Globus Toolkit conforms to the Generic Security Service API (GSS-API). This is a standardised API for security systems from the Internet Engineering Task Force (IETF) [WFK+04, WSF+03].

Resource Management

The Resource Management Architecture of the Globus Toolkit is a layered system. It contains high-level global resource management services on top of local resource allocation services. Figure 2.1 shows an overview of the components in the resource management architecture. Three main components can be observed:

- An extensible Resource Specification Language (RSL) [RSL05] which provides a method for exchanging information about resource requirements between all of the components in the Globus resource management architecture.
- An interface to all of the various local resource tools like LSF, NQE, LoadLeveler and Condor. That interface is provided by the Globus Resource Allocation Manager (GRAM) [ZKA04, Roy01].

- The co-allocation service is named Dynamically-Updated Request Online Coallocator (DUROC). It coordinates single requests that can span multiple GRAMs [CFK99].

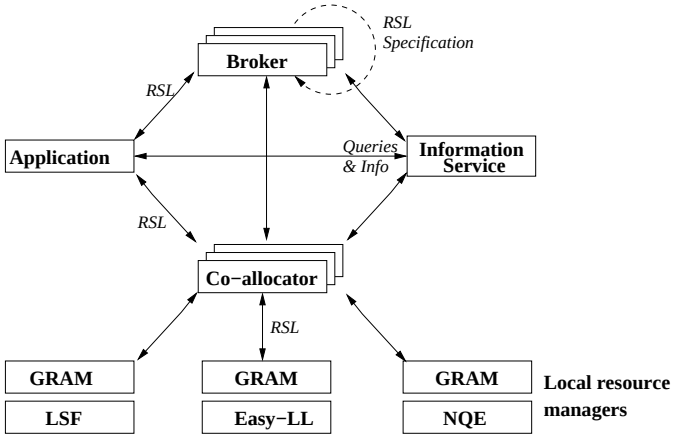


Figure 2.1: Globus Toolkit resource management architecture [Glo05]

Information Management

The information management is implemented by the Globus Metacomputing Directory Service (MDS). It provides the necessary tools to build an LDAP-based information infrastructure for computational GRIDs [Fit01, FvL98].

Data Management

A secure, high-performance and reliable data transfer protocol, that is optimised for high-bandwidth wide-area networks is Data Management implementation. It is called GridFTP and is based on FTP, but provides enhanced features required by data and also computational GRID projects [ABK+05].

In general, Globus is similar to a distributed operating system with uniform access to system features. Globus uses a standard application programming interface (API) for sending data and work to other machines using RSL, which is considered as a common notation for describing resource requirements.

2.1.5 GRID Layers

This sub-section discusses the layer of GRID architecture, which follows the "hour-glass" model as described by Foster [FKT01]. Broadly speaking, the GRID architecture identifies the fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another. Figure 2.2 defines a slim API for resource and connectivity protocols, so that collective services have a simple interface to work with; on fabric layer, many and often specialised resources are covered (e.g. storage, sensors), not just the usual for parallel computing such as memory, CPU etc. The fabric layer provides the resources that are shared by the GRID: CPU time, storage, sensors. The connectivity layer defines the core communication and authentication protocols required for GRID-specific multi-clusters transactions. In the resource layer there are *information protocols* that tells us about the state of the resource and *management protocols* that negotiate access to a resource. The collective layer includes directory services, scheduling, data replication services, workload management, col-laboratory services and monitoring services.

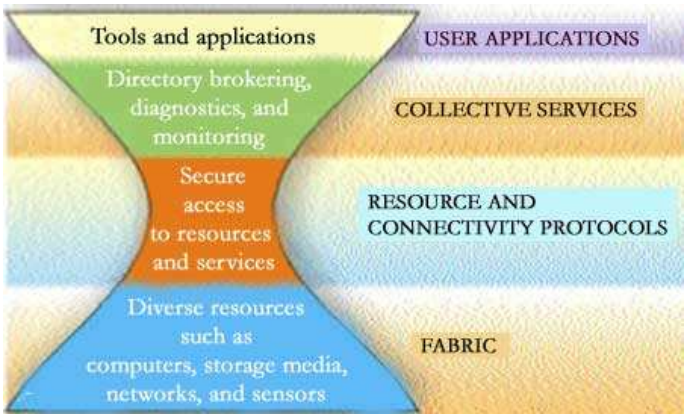


Figure 2.2: Layers in the Grid Architecture [FKT01]

2.1.6 Communication Libraries

The development of communication libraries influenced computation GRID [BFH03b]. During the 1980s and 1990s, application developers began to develop large-scale

systems that pushed against the resource limits of even the fastest parallel computers. Some groups began looking at distribution beyond the boundaries of the machine as a way of achieving results for problems of larger and larger size. As a conclusion of this, software for parallel computers focused on providing powerful mechanisms for managing communication between PEs and development and execution environment for parallel machine, and libraries like, Parallel Virtual Machine (Pvm), and Message Passing Interface (Mpi) developed to support communication for scalable applications [DFF+02].

Pvm

Pvm(*Parallel Virtual Machine*) emerged as one of the most popular cluster message-passing systems in 1992 [GBD+94]. Although Pvm did not originally work on nodes of multicomputers, more recently, multicomputer vendors have offered both layered and native versions of Pvm for multicomputer message passing [GKP96].

Mpi

The Mpi (*Message Passing Interface*) Standard defines a library of routines that implement the message passing model [GLS99]. These routines include point-to-point communication functions, where a *send* operation is used to initiate a data transfer between two concurrently executing program components, and a matching *receive* operation which is used to extract that data from system data structures into application memory space. It also provides collective operations such as broadcast and reduction that explicitly involve multiple PEs. In direct comparison, Mpi has a richer set of constructs than Pvm.

Mpich is a popular implementation of the Mpi standard [GLDS96]. It is a high-performance, highly portable library originally developed as a collaborative effort between Argonne National Laboratory and Mississippi State University. It was one of the first widely available Mpi implementations

Globus-enabled Mpi: Mpich-G

The Globus Toolkit enables access to various computational resources, but it can not itself provide a convenient way to use several resources simultaneously, as discussed in Section 2.1.4. A Globus-enable version of Mpich, Mpich-G [FK98a], is based on the Nexus Communication Library [FKT96]. The Nexus library was the basis for all communications in Mpich-G. It supports multiple protocols including a highly efficient TCP implementation and automatic data-type conversions between different architectures. However, the Nexus Library delivers a poor performance especially in intra-machine communication. For instance, in the case of sending or receiving data, the data has to be copied from the application buffer to the Nexus communication layer and vice versa. So there was a need for improvement, addressed by a new library Mpich-G2.

Globus-enabled Mpi: Mpich-G2

The implementation of Globus Toolkit2, the most stable one and the one used in this thesis, no longer uses the Nexus library. It is said to have re-implemented the 'good' parts of Nexus and improved the others [AHS$^+$]

The Globus-enable version of Mpich, Mpich-G2 [FK98a], supported by the Globus Toolkit since version 1.1.4 and conforms to the Mpi standard 1.1 with some additional features of Mpi 2.0. In Mpich-G2, data flows between applications' buffers without intermediate interfere from the communication layer, which improves Mpich-G2 performance by decreasing latency [KTF03].

There are new features of Mpich-G2 that do not exist on other Mpich implementations, for instance the ability to specify the IP address or network interfaces to be used, to specify IP port ranges if there are firewall issues and to tune the TCP buffer sizes. All of these can be done in RSL as part of the job requirements. In the case of computational Grid resources may be connected both by short distance and fast local network and by longer and wide area networks. Mpich-G2 separates communication methods into groups, each with similar properties based on their performance. In turn Mpich-G2 is able to maximize communication over

the fastest links and to minimise the communication over the slowest links.

2.2 Load Distribution

In computer systems in which the PEs possess localised memories and no shared global memory exists, processing work must be transported from heavily loaded PEs to lightly loaded PEs. The act of deciding upon what work should be moved where is termed *load distribution*. A major design issue for software designed to run on multi-clusters GRID architecture is the development of effective techniques for the distribution of threads amongst the PEs.

Load distribution is only one branch of a family of global scheduling techniques [CK88, CK95]. An alternative is *task assignment* which entails compile-time decision-making about the destination of a thread. Such algorithms can be derived by profiling, or heuristic prediction, but suffer from the unpredictability of general programming, as well as from the flaws of modelling and the inadequacies of heuristic identification.

The load distribution family of global scheduling techniques may be further subdivided into *load balancing* and *load scheduling* [CK88, CK95, Gos91]. In the former, an attempt is made to truly balance the load/utilisation of all PEs in the multicomputer; the latter sub-categorisation of load distribution algorithms requires no global balance, the goal is to equalise the load at a local level rather than a global level.

This thesis is concerned with load sharing in a heterogeneous environment in PE clusters levels and does not attempt to seek a global balance of load for all PEs in the computational GRID.

The taxonomy of Casavant and Kuhl further subdivides load distribution as shown in Figure 2.3 [CK88, CK95, SKH95, Der02]. However, before describing those subsystems, it is important to define thread and processes and the differences between them.

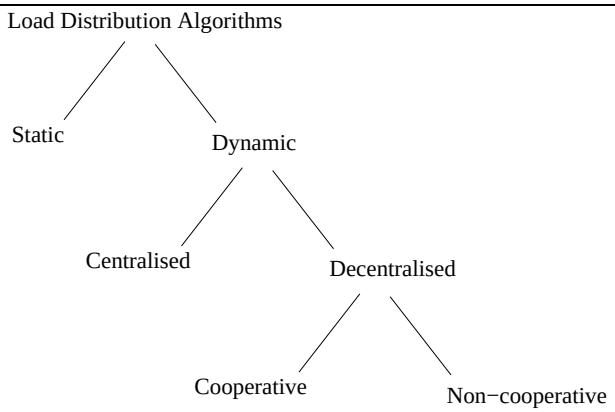


Figure 2.3: Load distribution algorithm classification

2.2.1 Threads Compared with Processes

Threads are distinguished from processes in that processes are typically independent, carry considerable state information and have separate address spaces, as described by Hammond and Michaelson in [HM99]. Multiple threads, on the other hand, typically share the state information of a single process, and share memory and other resources directly.

In general, threads are a way for a program to split itself into two or more simultaneously running tasks. In principle, multiple threads can be executed in parallel on many computer systems. This multithreading generally occurs by time slicing where a single processor switches between different threads or by multiprocessing where threads are executed on separate processors. Threads are similar to processes in this matter, but differ in the way that they share resources.

The related use of the term threads in this thesis is for threaded code, which is a form consisting entirely of subroutine calls, written without the subroutine call instruction, and processed by multiprocessing.

An advantage of a multi-threaded program is that it can operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines, or computational Grid (multi-clusters) environment. This is because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviours. In order for data to be correctly

manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require atomic operations in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks [Hyd, Wal].

Generally threads are implemented in one of two ways [LB96]: preemptive multithreading, or cooperative multithreading. Preemptive multithreading is generally considered the superior implementation, as it allows the system to determine when a context switch should occur. Cooperative multithreading, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing priority inversion or other bad effects which may be avoided by cooperative multithreading. The thread implementation presented in this thesis is classified under cooperative multithreading.

2.2.2 Static and Dynamic Load Distribution Subsystem

In static load distribution, PEs are assigned tasks at compile time, i.e. before the execution begins. Information regarding task execution times and processing resources is assumed to be known at compile time. Hence, a task is always executed on the processor to which it is assigned. However, the execution time of all of the tasks is not the same and over a period of time processing resources in a Grid network may change [SWP90]. Therefore, static load distribution is not appropriate for a computational Grid.

Static and dynamic load distribution algorithms are distinguished by the dynamic load distribution algorithm's use of current state; behaviour of static load distribution is pre-determined and the current load of a PE does not contribute in the load distribution decisions. In contrast, dynamic load distribution [SKS92]

assumes that limited knowledge about the processes and PEs is available *a priori*, and load distribution decisions are made during execution. The advantage of dynamic load management over static load management is that the system need not to be aware of the runtime behaviour of the applications before execution. For this thesis, dynamic load distribution is particularly useful due to the GRID architecture which consists of dynamic and hierarchal networks of workstations in which the primary performance is maximising utilisation of the processing power to minimise execution time of the applications.

2.2.3 Centralised and Decentralised Load Distribution Subsystem

Under centralised load distribution one PE is the designated the load distributor and all other tasks must keep this PE informed of their load. This activity not only incurs a high communication load but also concentrates the load on the links connecting the load distributor to the reminder of the network. In the context of this thesis where the GRID architecture is the target, centralised policy is very expensive due to the high latency between the PEs in the GRID network. An additional factor in centralised load distribution is the vulnerability of the central load distributing coordinator; failure of this PE results in termination of load distribution activity.

Often a better approach for removing the load processing bottleneck of the central load distributor is to decentralise load distribution and allow each PE to be responsible for its own load distribution activity. In the decentralised load distribution, each PE keeps its own local image of the system load.

2.2.4 Cooperative and Non-Cooperative Load Distribution Subsystem

Under a decentralised load distribution system, a PE may make choices about the transfer of threads on its own or through dialogue with other PEs. Under this dialogue, each PE passes its current load information to its neighbour at preset

time intervals resulting in the dispersement of load information among all PE in a short period of time and in a non-expensive way. A load distribution policy that utilises dialogue in the decision making process is said to be cooperative load distribution. The converse is termed a non-cooperative load distribution [Ros98].

2.2.5 Thread Placement and Thread Migration

Load distribution algorithms seek different characteristic in target threads. A load distribution algorithm may transfer threads after they have begun execution or before. The transfer of a thread after it has begun execution is called *preemptive transfer* or *thread migration*. The transfer of a thread prior to commencing its execution is called a *non-preemptive transfer* or *thread placement/allocation*. Transferring a thread that has begun executing is more complex than moving a thread yet to commence execution as the state of an evaluation, or partially evaluated, thread is typically relatively large and intertwined [Esk89].

GUM (see Section 2.4) is designed to support thread migration but at the beginning of this thesis thread allocation only had been presently implemented. Du Bois [DB05, DBLT02] has recently implemented thread migration transfer for GUM.

2.2.6 Mode of Operation in Dynamic Load Distribution

Dynamic load distribution algorithms can be separated into three classes depending upon the mode in which a task operates in order to facilitate load distribution [WM95, WM85]:

- sender-initiated algorithms,
- receiver-initiated algorithms,
- adaptive algorithms.

Sender-Initiated Load Distribution: *Sender-initiated* or *active* dynamic load distribution algorithms [ELZ85, Loi98], are those in which an over-loaded PE

searches for lightly loaded PE to which a thread may be transfered. Sender-initiated strategies outperform their receiver-initiated counterparts when overall system load is light to moderate. Although this gives a more even load distribution it may yield a deterioration in the performance if all PEs are equally loaded. Through the rest of this thesis we use active load distribution term to refer to sender-initiated load distribution.

Receiver-Initiated Load Distribution: *Receiver-initiated* or *passive* dynamic load distribution, which is also sometimes called *work stealing*: in this type of dynamic load distribution lightly loaded PEs have to explicitly ask for work from PEs with excess load. This strategy minimises the overhead during periods in which all PEs are busy anyway. GUM (see Chapter 2.4) is a system of passive load distribution, uses a "fishing" mechanism where requests are sent to a random PE. Through the rest of this thesis we use passive load distribution term to refer to receiver-initiated load distribution.

Adaptive Load Distribution: This the ideal case, a system that behaves in an active manner when overall system load is light and in a passive manner when overall system load is heavy. A system that alters its behaviour in such a manner is called an *adaptive* dynamic load distribution [ELZ85].

This thesis presents a semi-adaptive load distribution where PEs switch from passive manner to an active manner according to the load in different clusters as described in chapter 4.

2.2.7 GRID Scheduling

Although computational GRIDs falls within the distributed parallel computing domain, it has a lot of unique characteristics that make the scheduling in this environment highly difficult. A special load distribution mechanism should be implemented to overcome the challenging characteristics of computational GRID to deliver the potential high performance services [Ber99].

The grand challenges imposed by the computational GRID environment are examined in the following:

Resource Heterogeneity

Computational GRIDs mainly have two categories of resources: networks and computational resources. Heterogeneity exists in both of the two categories of resources. First, networks used to interconnect these computational resources may differ significantly in terms of their bandwidth and communication protocols. Second, computational resources are usually heterogeneous in that these resources may have different hardware, such as number of processor, physical memory, CPU speed and so on. The heterogeneity results in differing capability of processing jobs. Resources with different capability could not be considered uniformly. This thesis presents an adequate load distribution mechanism which addresses the heterogeneity and leverages the different computing power of diverse resources.

Resource Non-Dedication

Because of non-dedication of resources, resource usage contention is a major issue. Competition may exist for both computational resources and interconnection networks. This thesis targets dedicated computational resources, which leaves only the interconnection networks as non-dedicated. One consequence of the non-dedicated interconnection network is that behaviour and performance can vary over time. For example, in wide area networks, network characteristics such as latency may be varying over time. Due to this the new implemented load distribution monitors the latency continuously through the execution time to avoid high-latency communication.

Due to these challenging issues in the computational GRID environment, there have been a number of efforts attempting to design scheduling systems, each having its unique features.

Application Awareness

Application awareness can be divided into two levels. First, application-level scheduling which makes use of knowledge of applications as much as possible. Such kind of scheduling results in custom schedulers for each application attempting to maximise application performance, measured as runtime or speedup, with little regard to overall system performance. The complexity of application-level scheduling is the order of the applications considered. The best known example for application-level scheduling is AppLeS [BW97]. Second, resource-level scheduling, which does not use much knowledge of GRID applications. In this applications neither specify resource requirements nor provide application characteristics. Condor is an example which uses resource-level scheduling [BL99]. The new implementation presented in this thesis also uses resource-level scheduling.

Inter-Job Dependency

Given an application, the constituent jobs may be either dependent or independent. The scheduling mechanism for a set of independent jobs differs significantly from those for a set of dependent jobs. It is more complicated to schedule dependent jobs. Among existing GRID scheduling systems, some are intended to deal with independent jobs for simplifying the application model, for example, Nimrod-G [BAG00, AGK00] and Legion [GLFK98]. The scheduling mechanism presented in this thesis deals with dependent jobs.

Information Service

The scheduler determines the state information of all the resources in a computational GRID through the information service before making a scheduling decision. Different scheduling mechanisms adopt different methodologies to provide information service. There are three categories of information service:

- centralised, where there is a single entity that maintains the state information of all resources. Globus MDS [ZFS03] is one example using a centralised

scheme;

- decentralised, where every resource is responsible for maintaining its current state information locally, and answering queries from different resources. The decentralised scheme is suitable for large scale computational GRIDs, however the large overhead should be carefully considered. NWS [Wol98] uses the decentralised scheme;

- hybrid, under which scheme resources are aggregated into several groups. Within each group, the centralised scheme is applied, and over the groups, the decentralised scheme is applied.

In this thesis, we have adopted a decentralised mechanism for information service.

Scheduler Organisation

The GRID scheduler organisation can be also classified as: centralised, decentralised, and hierarchical.

In the centralised scheme, all tasks are sent to the centralised scheduler where there is a queue for holding all the pending tasks. The decentralised scheme is not very scalable with increasing number of resources. The central scheduler may prove to be a bottleneck in some situation. In contrast, the decentralised scheme distributes responsibility to every site (PE). Each site in the computational GRID acts as both a scheduler and a computational resource. The tasks are submitted to the local GRID scheduler where the task originated. Since the responsibility of the scheduling is distributed, the failure of a single scheduler does not effect others' scheduling. In the hierarchical scheme, the scheduling process is shared by different levels of schedulers. Resources here are organised in hierarchical way, where the higher level resources manage the scheduling for the resources in the lower level. Compared with the centralised scheduling, hierarchical scheduling addresses the problem of single-point-of-failure. Nevertheless, it also retains some of the advantages of the centralised scheme.

This thesis presents a decentralised scheduling organisation.

Rescheduling

Rescheduling is an important technique used to enhance system reliability and flexibility. GRID scheduling systems can be classified into two categories: one with rescheduling support and the other without re-scheduling.

The load distribution mechanism presented in this thesis supports rescheduling function.

2.2.8 Summary

Table 2.1 [ZHU02] compares the most common scheduling system on the GRID with the new adaptive load distribution presented in this thesis, *GRID-GUM2*.

System	Developer & Year	Resources	Applications	Scheduling
Condor [BL99]	University of Wisconsin, Madison (1988)	single-domain GRID, non-dedicated, non-time-shared	single-job application	centralised info, decentralised scheduler, rescheduling-support
Condor-G [FTL+01]	University of Wisconsin, Madison (2001)	multi-domain GRID, non-dedicated, non-time-shared	single-job application	centralised info, decentralised scheduler, rescheduling-support
AppLes [BW97]	University of California, Sand Diago(1996)	single-domain GRID, non-dedicated, time-shared	diverse applications	decentralised info, decentralised, scheduler, rescheduling-support
Legion [GLFK98]	University of Virginia (1998)	single-domain GRID, non-dedicated, time-shared	diverse applications	centralised info, decentralised scheduler, rescheduling-support
Nimrod/G [AGK00]	Monash University Australia (2000)	multi-domain GRID, non-dedicated, time-shared	soft Real-time parametric study	centralised info, decentralised scheduler, non-rescheduling-support
GRID-GUM2	Heriot-Watt University (2004)	multi-domain GRID, dedicated, non-time-shared	single-job application	decentralised info, decentralised scheduler, rescheduling-support

Table 2.1: Comparison of Scheduling Systems

2.3 Functional Languages

Functional languages are general purpose programming languages supporting programming at a higher level of abstraction than conventional imperative languages like C [KR88] and Fortran [CW88]. Functional programming languages express computation in terms of pure functions. A program is expressed as a function from its input to its output. These languages are usually describe as *declarative* languages which involves specifying 'only' what is to be computed while imperative programming is prescriptive, specifying also the details the computation steps.

Perhaps the most important advantage they have, as described by Hughes [Hug89], are their powerful facilities for modular design. In particular higher order functions enable common patterns of computation to be captured. This may be at a relatively low level such as a function for applying another function elementwise across a data structure or it may be the abstraction of a whole algorithm. Conventional imperative languages do not usually include such powerful abstraction facilities. For example in languages like Pascal it is not possible to write generic list processing functions. This is due to limitation of the type system and limitation of procedural abstraction. Conventional languages are also more limited in the kind of abstractions which may be defined and used. The better the abstraction facilities a language offers, the more ways there are of breaking up and hence solving a problem. Abstraction facilities are the key to modularisation and hence to programming in the large. Thus functional languages are good for programming in the large.

According to Roe [Roe91], there are at least two other benefits of functional languages. The first is that there are mathematically tractable and hence they can be reasoned about more easily than conventional languages. This is also makes program derivation much easier. The second benefit is that functional programs are amenable to parallel evaluation. This is one of the core of this thesis.

Examples of functional programming languages includes Haskell [PHA+99], Miranda [Tur85, Tur86] and ML [AJ90]. There are also object-oriented functional

languages, for example Clover [CB97, BC97] and Object Caml [LDG+01].

2.3.1 Parallelism or Concurrency

The terms *concurrency* and *parallelism* have sometimes been used interchangeably in the past. Both types of system involve executing processes or tasks at the same time. Modern usage [BA90] is, however, to use the term "concurrency" for systems which involve a number of independent, but collaborating, processes, such as graphical user interfaces or operating systems. The term "parallelism" is used for systems involving the cooperation of a number of inter-dependent tasks on a single activity.

The purpose of concurrency is to support abstraction and to improve security by separating activities which are logically independent processes, but which may take place simultaneously. The purpose of parallelism in contrast, is to improve performance, usually in terms of speed, throughput or response time by creating subtasks to deal with units of work [HM99].

In this thesis we are explicitly concerned about parallelism rather than concurrency.

2.3.2 Classification

Functional languages are often classified, on the basis of their semantics, into strict, non-strict and lenient [Loo99]. A function is *strict* in an argument x if, whenever the value of x is undefined, the result of the function is also undefined. A strict function is a partial function which is strict in at least one of its arguments. A *non-strict* function is a partial function that may be defined even when one of it arguments is not defined. Strict functional languages are therefore those that support strict functions while non-strict languages are those that support non-strict functions. *Lenient* languages combine the features of both strict and non-strict languages: they support functions which can return results even when their computation may not terminate.

Lazy evaluation [PJCSH87, Par91, JCS89] is the implementation technique often used to implement non-strict semantics. Lazy evaluation starts evaluating the function's arguments as and only when they are used. Lenient evaluation starts the evaluation of the function in parallel with the evaluation of all the arguments of the function. Lazy evaluation enables functional languages to express algorithms involving potentially infinite data structures succinctly. Such algorithms are awkward to express in a language without lazy evaluation [Cli82].

Also laziness is an implementation technique that encompasses both normal order evaluation (in that it pursues a leftmost outermost reduction strategy to achieve weak head normal forms) and sharing (in that β-reduction is achieved using call by need rather than call by name); the later is necessary so that for example, in the expression $\lambda x.(x + x)$ the argument x is only evaluated once. In practice, the normal order evaluation of function arguments also extends to data constructors so that, for example, $(1/0) : []$ does not evaluate $(1/0)$ unless, and until, the head of the list is demanded [Cla99, MHC99].

2.3.3 Parallel Functional Language Mechanisms

Parallel Graph Reduction

An important evaluation mechanism for non-strict functional language is *graph reduction* [Bar84]. One method for implementing the graph reduction data structure is to translate the program to combinators [CF58]. A key feature of this method is that all free variables are abstracted from each function in the program. The program is represented as a computation graph, with instances of variables replaced by pointers to subgraphs which compute values. Graphs are evaluated by repeatedly applying graph transformations until the graph is irreducible. The irreducible final graph is the result of the computation.

According to Loidl [LTB01, Loi98], graph reduction has several advantages:

- It is easy to express sharing of program expressions by sharing in the graph;

- a call-by-need evaluation can be easily implemented by overwriting the reduced node with its result;
- independent part of the graph can be evaluated in parallel.

From the parallel point of view the most important advantage of graph reduction is the ease of expression of parallel computation. A parallel graph reduction model can be very naturally expressed as a spark pool, i.e. a pool consisting of pointers to unevaluated expressions ("thunks"), and a set of processors that take sparks out of this pool and execute them by creating a `thread`, and independent process performing standard graph reduction. These threads are kept and maintained in a separate thread pool [AJ89, Loi98].

Skeletons

Skeletons or skeleton-based approaches define a set of parallel templates [Col89]. They are a popular parallel coordination construct. Typically, a language has a small set of predefined skeletons, where each skeleton is a higher-order function describing a common coordination pattern with an efficient parallel implementation [Col99].

The programmer writes the program using skeletons as appropriate. A parallelising compiler can then exploit the rules provided for each skeleton, in order to produce an efficient parallel implementation of the program on the target architecture.

From the functional programmer's perspective, a skeleton is simply a normal higher-order function. Each higher-order function is mapped to a different abstract parallel process topology, with parameters specifying details of the tasks that are to be performed.

Examples of well-developed systems using a skeleton-based approach for parallelism are SCL [DGT96] and P3L [BDO+95]. Both systems define a coordination language that can be freely combined with an arbitrary computation language. In practice these systems often use C or FORTRAN as computation languages. As

a crucial technique for the development of larger applications these languages allow the specification of data re-distribution to compose skeletons with conflicting data distributions.

2.3.4 Parallel Functional Programming

Popular general-purpose languages lack parallelism primitives as part of the language. Most often this factor necessitate the redesign of the algorithm and complicate its translation to a program. Moreover, some object-oriented languages, e.g. Simula [ND78] have no provision for parallelism, others, e.g. C++ [Str85], have library extensions but needs explicit user specification. Ada [Geh84] and Java [AG96] offer powerful parallelism primitives in the language but these are constrained by the scourge of imperative programming: the management of state. Each parallel task executes with shared variables. Consequently each task must manage shared access to variables and synchronisation throughout the execution which includes task termination.

Functional programming languages lack state, are referentially transparent, and inherently possess many opportunities for parallel evaluation [Bac78].

In principle no syntactic constructs are necessary to identify parallelism in a functional program, and no complex mechanisms are required for synchronisation. As functional programs are free from side-effects, a sub-expression can be executed in any order since its execution cannot affect the value of any other sub-expression of the expression(s) in which it occurs.

Parallel implementation of functional languages continue to receive increasing attention by researchers. Functional programming languages are arguably the most suited to computational Grid hardware platforms because of their clean semantics [Bar84, PJCSH87]. The programmer needs only to express the algorithm in executable form and the parallelism can be inferred in array/list element calculation, function argument evaluation, conditional expression evaluation, evaluation of operands to operators, or sub-expression evaluation.

In short, the goal of parallel programming is to achieve higher performance,

by reducing runtime. So the advantages of parallel programming with functional languages can be summarised as follow:

- Functional programs designed for parallel evaluation may be reasoned about in the same way as sequential programs.
- Parallel functional programs, unlike other parallel programs, need no communication, synchronisation or mutual exclusion to be specified explicitly. This all occurs implicitly in the program graph.
- Deadlock can only arise when the result of a program is undefined.

In a parallel setting, side-effects are an anathema to automatic or semi-automatic parallelisation since they inhibit easy program decomposition into parallel tasks and introduce new dependencies between tasks which can be difficult or impossible to disentangle without using explicit parallel control.

Because a purely functional language has no side-effects, it is relatively easy to partition programs so that sub-programs can be executed in parallel. Any computation which is needed to produce the result of the program may be run as separate task. The control dependencies which are implicit in the language serve to enforce any sequential behaviour, and also to limit the creation of excess parallelism some extent.

2.3.5 Parallel Control

The two extremes of parallel control are typified by fully explicit approaches, where all behavioural details are specified, including parallel partitioning, task and data distribution, load management and communication, and fully implicit fully approaches where the compiler make all such decisions. In between lies a wide range of semi-implicit/explicit approaches.

Purely Implicit Approaches

The most implicit approaches, requiring least programmer input are exemplified by pH [NPA92]. pH is an implicitly parallel language based on Haskell. The

arguments to a function are evaluated in parallel, and each iteration of the parallel loop-construct is similarly executed as a separate task.

Restricted Implicit Approaches

Restricted implicit approaches match certain language characteristics to desirable program properties. For example in the skeleton approach [Col99] certain patterns of computation are recognised and matched with suitable templates of parallel behaviour.

Controlled Parallelism

Annotation-based approaches may fall either side of the implicit/explicit divide. If an annotation is a directive to the compiler, then this is clearly example of explicit parallelism. If the annotation is a suggestion, however, that may perhaps be checked by the compiler or even ignored entirely, then the construct lies more in the realm of implicit parallelism. This is not merely a technical distinction: in implicit parallel systems, overall control lies in the hand of the compiler and runtime systems (which is automatic parallelism), whereas in an explicitly parallel system overall control rests squarely in the hands of the programmer (which manual parallelism)

Other controlled approaches include the evaluation strategies [THLP98] and first class schedules [Mis94]. In these systems, functions are higher-order functions that manipulate sequential or parallel program components to yield a more complex parallel program behaviour, but whose definition is entirely within the normal semantics of the sequential programming language. We refer to this approach in this thesis as *semi-implicit* parallelism.

Explicit Approaches

In the explicit approach not only is every detail of parallel execution under the programmer's control, but it must be specified in the parallel program. In principle, this allows a skilled programmer to produced a highly optimised parallel program for some architecture. This is usually achieved by providing new parallel control constructs to deal with parallel partitioning, communication, and task placement, etc. The typical examples for explicit approach in functional language are Caml with the Mpi message passing library and Concurrent ML [Ser99, Nie99].

Semi-implicit parallel languages provide a few high-level constructs for controlling key coordination aspects, while automatically managing most coordination aspects statically or dynamically. Historically annotations were commonly used for semi-implicit coordination, but more recent languages provide compositional language constructs. As a result, the distinction between semi-implicit coordination and coordination languages is now rather blurred, but the key difference in the approach is that semi-implicit language aim for minimal explicit coordination. Due to this we proposed in this thesis to use a language with semi-implicit parallel coordination, GpH.

2.3.6 Parallel Functional Language

This section compares the three parallel functional languages PMLS, GpH, and Eden. The three languages have been chosen for the following reasons. Firstly to be consistent with a high-level computation language we selected languages with high level coordination and exclude languages with imperative or low-level coordination. Secondly the languages represent a range of language designs, e.g. both eager and lazy languages, and with coordination ranging from almost entirely implicit (PMLS) to a language (Eden) in which processes can be manipulated by the programmer. Thirdly the languages represent a range of implementation designs, e.g. both those with predominantly static coordination (PMLS) and those with

predominantly dynamic (GpH). Finally we have selected three of the relatively few robust parallel functional languages available.

PMLS

Parallel ML with Skeletons (PMLS) is a parallelising compiler for the full purely functional subset of Standard ML, that realises parallelism in higher-order functions as algorithmic skeletons [MSBK01]. The PMLS system is based on skeletons that seek to minimise programmer involvement in identifying and exploiting parallelism.

Since the only parallel constructions that are available to the programmer are the higher-order functions that have been provided by the language, programmers must design parallel algorithms by adapting the sequential source to these functions. The compiler and runtime system are jointly responsible for setting up the corresponding process topologies, and for mapping processes to processors.

Higher-order functions may be given different behavioural interpretations when compiling for different target architectures. This allows a single higher-order functions to abstract over a range of possible parallel behaviours, which are selected on the basis of concrete details such as communication latency, or the granularity of the tasks to which the function is applied. In essence, skeletons modify behaviours but not values.

The PMLS compiler generates parallel code solely from calls to `map` and `fold`. No other SML constructs are provided or exploited for parallelism. However, the system enables the introduction of new higher-order functions with new skeletons.

GpH

GpH [THLP98] is a modest conservative extension of Haskell98 [PHA+99] realising a thread-based approach to parallelism. *Thread-based approaches* to parallelism allow parallel threads to be created, but do not provide mechanisms to control those threads. Threads are thus managed entirely under runtime-system control. By combining simple thread primitives with higher-order functions, high-level abstractions can be constructed, such as the evaluation strategy

approach [THLP98].

GpH provides parallel (`par`) and sequential (`seq`) composition as coordination primitives. Denotationally, both compositions are projections onto the second argument. Operationally `seq` causes the first argument to be evaluated before the second and `par` indicates that the first argument may be executed in parallel. The latter operation is called the "sparking" of parallelism and is used in different variants in many parallel languages. The runtime-system, however, is free to ignore any available parallelism. In this model the programmer only has to expose expressions in the program that can usefully be evaluated in parallel. The runtime-system manages the details of the parallel execution such as thread creation of communication.

Experience of implementing non-trivial programs in GpH shows that the unstructured use of `par` and `seq` can lead to rather obscure programs. This problem can be overcome with *evaluation strategies* [THLP98]: lazy, polymorphic, higher-order functions controlling the evaluation degree and the parallelism of a Haskell expression. Evaluation strategies provide a clean separation between coordination and computation. The driving philosophy is that it should be possible to understand the computation specified by a function without considering its coordination.

GpH programs are developed with an integrated suite of sequential and parallel software tools, based on the Glasgow Haskell Compiler (GHC) [PHH+93]. The tools for sequential software development include: the Hugs interpreter, for fast development, the GHC compiler and sequential runtime system for optimising compilation to sequential code; and sequential time and space profilers integrated into GHC [SP95]. The tools for parallel software development include: the GranSim parameterisable parallel simulator [HLP95] for flexible and accurate simulation of the parallel behaviour on a range of parallel machines; the GHC compiler and GUM parallel runtime system for parallel execution on multiprocessors; a set of visualisation tools for both GranSim and Gum, visualising the activity of a parallel machine in several levels of detail; prototypes of more detailed parallel profilers [KHT98].

Eden

Eden [BLOP96] extends the lazy functional language Haskell by syntactic constructs to explicitly define and instantiate processes. In contrast to the previous techniques, *process-based approaches* like Eden expose parallel tasks at the language level. The programmer must then manage the tasks using the control mechanisms provided in the language. Eden is explicit about process creation and about the communication topology, but implicit about other control issues such as sending and receiving messages, and process placement. Granularity is under the programmer's control because he/she decides which expressions must be evaluated as parallel processes, and also some control of the load balancing is possible at the programmer's level.

Like GpH Eden is based on the Glasgow Haskell Compiler, and can use the same sequential profiling utilities. For parallel profiling Eden provides a simulator called Paradise [HPR00] which is based on GranSim, so that tuning the performance of an Eden program is a similar process to that in GpH.

Parallel programming in Eden can be done by explicitly defining and instantiating a process topology. This would be equivalent to sequential functional programming with explicit recursion. Sometimes this is appropriate, but an experienced functional programmer will try to use higher-order functions, i.e. skeletons, as much as possible in order to reduce the amount of work and the possibility of making mistakes. In a complex application both methods may be simultaneously needed. The main difference between Eden and more traditional skeleton-based languages, such as PMLS, is the fact that skeletons can be specified within Eden itself. Thus, Eden serves both as a computation and coordination language, providing a high degree of flexibility for the programmer.

2.3.7 Parallel Functional Language Implementation

PMLS: PMLS [MSBK01, TLP02] is an automatically parallelising compiler for a pure subset of SML. The execution costs of functions are profiled by executing a structural operational semantics. Based on this information a cost model for the

available skeletons, possibly nested, is used to select a decomposition and mapping of parallel threads. Measurements on a range of parallel machines including a Beowulf cluster exhibit good speedups for programs such as matrix multiplication, a ray tracer and a linear system solver [SMH01].

HDC: HDC [HL00] is a strictly-evaluated subset of Haskell with skeleton-based coordination. HDC programs are compiled using a set of skeletons for common higher-order functions, like fold and map and several forms of divide-and-conquer. Unlikely Haskell, HDC does not implement type classes, and has strict semantics to facilitate static thread placement. In summary, HDC has purely implicit threads with implicit interaction. It is location independent, since parallelism is not explicit in the program at all.

In HDC it is possible to achieve parallel execution of the program without any changes. In tuning the performance of the parallel program, however it is necessary to modify the code, so as to weaken data dependencies or to increase granularity.

A particular focus of the HDC system is the time and space efficient static thread placement. The compiler uses a library of skeletons to decompose a program into parallel threads and place the threads on the available PEs. In contrast languages such as GpH and Eden, use more flexible, but also more expensive, dynamic resource management.

DREAM: DREAM [BLOP97, LRS+03] is the parallel abstract graph-reduction machine implementing Eden and is largely similar to GUM. The main difference between GUM and DREAM is that in DREAM, the concept of a virtual shared heap does not exist. In general, the main bottlenecks in Eden are due to the packing and unpacking routines, which are not yet optimised. Moreover, as there are not yet multicasting facilities in Eden, once a packet has been sent to a process, the packing effort could be used in order to send the same packet to other process.

GUM: GUM (Graph reduction for a Unified Machine model) is a portable, parallel implementation of GPH and Eden [THM+96]. GUM was the first publicly-released parallel implementation of any functional language. GUM was designed and built to make parallel graph reduction more accessible to the wider community, and of more practical use.

GUM uses an abstract message passing implementation, originally built around the widely available PVM communication harness. In GUM, the units of computation are called threads. Each thread evaluates an expression to weak head normal form. GUM uses a graph reduction mechanism and in order to transfer a subgraph from one PE to another, GUM uses sophisticated packing and unpacking algorithms, which guarantee that all the links back to the original graph are maintained and that the duplication of work is avoided. GUM uses a passive work distribution scheme, where PEs looking for work send out requests for work [Cla99].

To synchronise multiple threads in GUM, a thread locks the node of the graph when starting its evaluation, and other threads requesting that data will be added to a blocking queue attached to the locked closure. Access to remote closures is managed by new FetchMessage nodes, representing global indirections. On requesting the contents of such a node a message will be sent to the target processor and the requesting thread will be added to a blocking queue [LRS+03].

2.4 GUM - A Parallel Haskell Runtime

2.4.1 Introduction

GUM (Graph reduction for a Unified Machine model) [HMP+95, THM+96] is a portable, parallel implementation of the non-strict purely-functional programming language Haskell. The Haskell compiler, in particularly GHC, in actual fact, many compilers in one. By providing command line options when compiling different facilities are enabled/disabled and different behaviours are exhibited. When the *-parallel* option is provided on GHC command line, execution involves

a runtime system that enables GUM.

GUM implements graph reduction using an abstract machine modelled on GRIP [HP90, HP92]. GRIP consists of two types of processor: Processing Elements (PEs) which perform graph reduction, and Intelligent Memory Units (IMUs) which hold the shared graph and manage global thread pool.

Although GUM had its inception on the specialised GRIP architecture, GUM is now architecture neutral and portable. Earlier redesign of the GRIP reduction system retained the IMUs, but in GUM these are removed by distributing the global memory amongst all PEs, rather than IMUs, using a globalised address space. Each PE has a local heap, which is independently garbage collected the collection of all local heaps provides a virtual global heap.

This section describes the design of the GUM 4.06. The key concepts in the design of GUM can identify in the following components:

- The *initialisation and termination* is responsible for controlling startup and termination, Section 2.4.2.
- The *thread management* is responsible for deciding when to generate a new thread and how to schedule the threads, Section 2.4.3.
- The *load distribution* is responsible for distributing the load in the parallel system so that idle time of PEs is minimised, Section 2.4.4.
- The *memory management* is responsible for controlling access to remote data and in GUM it implements a virtual shared heap, Section 2.4.5.
- The *communication* is responsible for transferring data and work between PEs, Section 2.4.6.

The materials presented in this section have been summarised from [THM+96, HMP+95, Loi02b, Loi02a, Loi98, Der02, LH96]

2.4.2 Initialisation and Termination

The first action of a parallel Haskell program in GUM is to create a PVM manager task, whose job is to control startup and termination. This manager task then

spawns the required number of logical PEs as PVM tasks, which PVM maps to the available processors. In this thesis, GUM has been adapted to run with MPI in particular MPICH and MPICH-G2, as a result of this the manager task job has been passed to certain MPI routines, as discussed in Section 3.2.1. Each PE task then initialises itself: processing runtime arguments, allocating heap etc. Once all PE tasks have initialised, and been informed of each others identity, one of the PE-tasks is nominated as the *main PE*. The main PE then begins executing the main thread of the Haskell program.

The program terminates when either the main thread completes, or encounters an error. In either case a FINISH message is sent to the manager task, which in turn broadcasts a FINISH message to all of the PE tasks. The manager waits for each PE task to respond before terminating the program.

During execution each PE executes the following scheduling loop until it receives a FINISH message.

Main Scheduler:

1. Perform local garbage collection, if necessary.
2. Process any messages
3. If there are sparks or runnable threads then call the thread scheduler. Otherwise send a FISH message requesting work, and block awaiting incoming messages.

The inter-PE message protocol is completely asynchronous. When a PE sends a message it does not await a reply; instead it simply continues or returns to the main scheduler.

2.4.3 Thread Management

In GUM a thread is a virtual processor. It is represented by a (heap-allocated) Thread State Object (TSO) containing slots for the thread's registers. As the

thread's stack grows further Stack Objects are allocated and chained on to the earlier ones.

Threads in GUM may be in any of the five states during execution:

- running,
- runnable: waiting to be scheduled,
- blocked: waiting for another thread to be complete,
- fetching: waiting for a value to arrive from a remote PE, or
- migrating: moving a thread form a busy PE to an idle PE.

Each PE has a pool of runnable threads, or rather TSOs, called its runnable pool, which is consulted in step (3) of the scheduling loop given earlier. The version of GUM used in this thesis does not support thread migration. Once a thread has begun execution on a PE it cannot be moved to another PE.

When thread is chosen for execution it is run non-preemptively until either space is exhausted, the thread blocks (either on another thread or accessing remote data), or the thread completes. Compared with fair scheduling, this has the advantage of tending to decrease both space usage and overall run-time [BRS94], at the cost of making parallel and speculative execution rather harder.

Sparks

Parallelism is initiated explicitly in a GPH program by the *par* combinator. The *par* combinator implements a form of parallel composition. Operationally, when the expression `x 'par' e` is evaluated, the heap object referred to by the variable x is *sparked*, and then e is evaluated. Quite a common idiom (though by no means the only way of using par) is to write

```
let x = f a b in x 'par' e
```

where e mentions x. Here, a *thunk* representing the call `f a b` is allocated by the `let` and then sparked by the par. It may thus be evaluated in parallel with e.

Sparking a thunk is relatively cheap operation, consisting only of adding a pointer to the thunk to the PE's *spark pool*. A spark is an indication that a thunk might usefully be evaluated in parallel, not that it must be evaluated in parallel. Sparks may freely be discarded if they become too numerous.

Synchronisation

It is obviously desirable to prevent two threads from evaluating the same thunks simultaneously, lest the work of doing so be duplicated. This synchronisation is achieved as follows:

1. When a thread enters (starts to evaluate) a thunk, it overwrites the thunk with a *black hole*. In fact, thunks are only overwritten with black holes when a thread context switches. The advantage of this `lazy black-holing` is that many thunks may have been entered and updated without ever being black-holed.

2. When a thread enters a black hole, it saves its state in its TSO, attaches its TSO to the queue of threads blocked on the black hole (the black hole's *blocking queue*), and enters the scheduler.

3. When a thread completes the evaluation of a thunk, it overwrites the latter with its value (the *update operation*). When it does so, it moves any queued TSOs to the runnable pool.

Notice that synchronisation costs are only incurred if two threads actually collide. In particular, if a thread sparks a sub-expression, and then subsequently evaluates that sub-expression before the spark has been into a thread and scheduled, then no synchronisation cost is incurred. In effect the putative child thread is dynamically in-lined back into the parent, and the spark becomes an orphan.

2.4.4 Load Distribution

GUM's load distribution mechanism has been designed to work in a flat architecture with uniform PE speed and communication latency. One of the key elements

of this thesis is to improve GUM's load distribution mechanism to work in a computational GRIDs environment that are hierarchical, heterogeneous and shared. GUM's load distribution works as follows.

If (and only if) a PE has nothing else to do, it tries to schedule a spark from its spark pool, if there is one. The spark may by now have been an orphan, because the thunk to which it refers may by now be evaluated, or be under evaluation by another thread. If so, the PE simply discards the spark and tries the next spark in first-in first-out (FIFO) order. If the PE finds a useful spark, it turns it into a thread by allocating a fresh TSO, and starts executing it.

GUM has a simple way of controlling the load distribution by specifying a hard limit on the total number of live threads, i.e. runnable or blocked threads. This avoids activating a huge number of sparks in the rather common case, where a newly activated thread needs remote data early on in its computation, therefore blocking quickly. This way of specifying a limit on the total number of live threads helps to improve the performance with some programs in a heterogeneous multicluster environment, as will be discussed in Section 5.4.1.

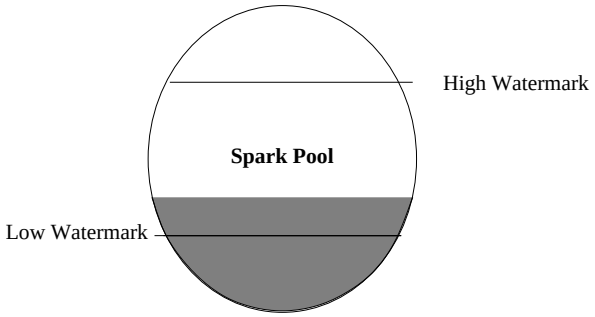


Figure 2.4: Low- and High-watermark mechanisms for load distribution in GUM.

If there are no local sparks, then the PE seeks work from other PEs by launching a FISH message that "swims" from PE to PE looking for available work. Initially only the main PE is busy - has a runnable thread - and all other PEs start fishing for work as soon as they begin execution. To provide better control of the distribution of sparks, low- and high-watermarks for the spark pool have been designed [Loi02a]. In this model the load distribution mechanism depends on the emptiness of the spark pool as sketched in Figure 2.4. The *low-watermark*

indicates how many sparks should always be kept local on a PE. If the number of sparks falls below this mark, no sparks will be exported and the PE will try to obtain new sparks from other PEs. The *high-watermark* indicates the maximum number of sparks that should be held in a spark pool. If the number of sparks exceeds this limit, the PE will start to off-load sparks to other processors without being asked for work. The high-watermark has not been implemented in GUM 4.06, and the low-watermark has not been activated in this thesis as it has little impact on performance [Loi01b].

When a FISH message is created, it is sent at random to some other PE. If the recipient has no useful sparks, it increases the "age" of the FISH, and sends the FISH to another PE, again chosen at random. The "age" of a FISH limits the number of PEs that a FISH visits: having exceeded this limit, the last PE visited returns the unsuccessful FISH to the originating PE. On receipt of its own, starved, FISH the originating PE then delays briefly before lunching another FISH. The purpose of the delay is to avoid swamping the machine with FISH messages when there are only a few busy PEs. A PE only ever has a single FISH outstanding.

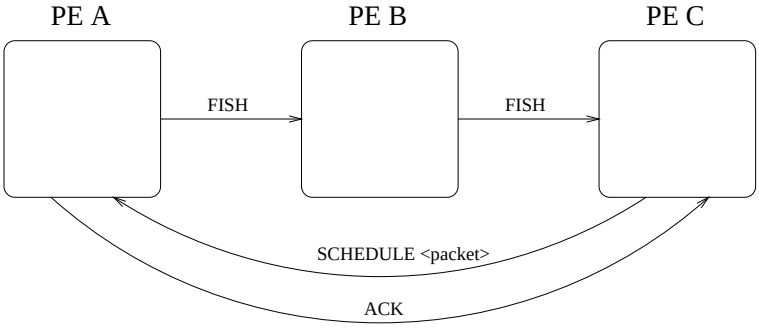


Figure 2.5: Fish/Schedule/Ack Sequence

If the PE that receives a FISH has a useful spark, it sends a SCHEDULE message to the PE that originated the FISH, containing the sparked thunk packaged with near by graph. The originating PE unpacks the graph, and adds the newly acquired thunk to its local spark pool. An ACK message is then sent to record the new location of the thunk sent in the SCHEDULE. Note that the originating PE may no longer be idle because, before the SCHEDULE arrives, another incoming

message may have unblocked some thread. A sequence of messages initiated by a FISH is shown in Figure 2.5.

2.4.5 Memory Management

Parallel graph reduction proceeds on a shared program/data graph, so a primary function of GUM is to manage the virtual shared memory in which the graph resides. GUM uses a *flat memory hierarchy*, i.e. the access to any closure in one PE's heap is uniform. Every globally visible closure in the heap is identified via a global address (GA), a globally unique identifier. Global indirection closures (FetchMe) use the GA to identify the remote object. The mapping of GAs to local heap addresses and vice versa is done via a hash table, the Global Indirection Table (GIT), Figure 2.6.

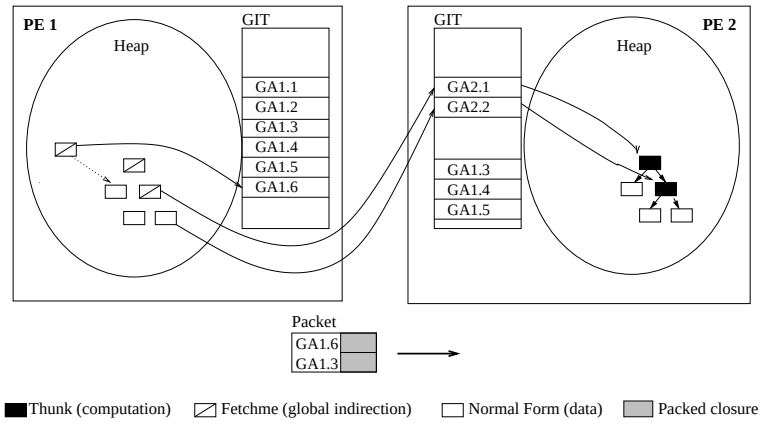


Figure 2.6: Transfer of graph structures

Figure 2.6 elaborates on the allocation of GAs and the transfer of graph structures on two PEs. This snapshot shows the heaps on two PEs after having completed the transfer of the five closure graph with root `GA2.1` on PE2 (originally `GA1.1` on PE1).

This design enables separate local Garbage Collection (GC) on each PE, provided that the GIT is rebuilt after every GC to map all live GAs to their new addresses in the heap.

Note, that the mapping of global to local addresses is needed for determining whether a copy of a newly imported graph structure already exists on that PE. In

this case, the less evaluated version of the graph will become an indirection to the evaluated version. This avoids duplicating data that might have been imported via different PEs.

2.4.6 Communication

GUM utilises PVM for communication via message passing and thus obtaining portability. However, GUM is designed to be independent of the communication library, and there are two aspects to this independence. The first is that GUM only uses a small number of common communication patterns. Moreover, only point-to-point communication is used during execution of the GPH program; broadcast and barrier synchronisation are also used, but only during initialisation and termination, and hence an effective implementation is not essential. The second aspect of communication library independent is that a layered architecture is used to isolate the use of communication routines to just 3 of the 150 modules in the runtime system. This thesis exploited the GUM's architecture-independence from any specific communication libraries to construct different implementation of GUM that uses MPICH and MPICH-g2, as outlined in Chapter 3.

2.5 Summary

Computational GRIDs, as presented early in this Chapter, potentially offer cheap large-scale high-performance systems, but are a very challenging architecture, being heterogeneous, shared and hierarchical. Rather than requiring a programmer to explicitly manage this complex environment, this thesis aims to program computational GRIDs using a language with high-level parallel coordination that hides the complexities of both the GRID infrastructure and the underlying hierarchical, heterogeneous and shared architecture. This thesis use the parallel functional language, GPH, with largely automatic management of parallel coordination.

GPH is currently used on classical HPCs and abstracts from low level coordination issues such as work and data distribution, and both thread communication

and synchronisation. Moreover, GpH is an extension of the GHC sequential implementation [Pey96] and is robust, publicly-available, and arguably the leading parallel functional language implementation [LRS+03]. The high-level coordination in GpH is supported by a runtime environment, Gum. We view Gum as an ideal platform to adapt to the heterogeneous high latency Grid environment as it supports architecture independence, is readily ported and is implemented on both high-latency Beowulf clusters and low-latency SUNServer SMPs [TL+00]. Gum's effectiveness has been demonstrated by parallelising numerous large programs with a relatively small programming effort, achieving wall-clock speedups over the equivalent optimised sequential programs.

This thesis presents *GRID-GUM2*, which is the distributed virtual shared-memory implementation of GpH for computational Grids.

The system most closely related to our philosophy of semi-implicit management of parallelism in a high level language is the ConCert system [Con04] and the Hemlock compiler [Mur03], which translates a subset of ML to machine code, for execution on a Grid architecture. In contrast to our work, parallelism is expressed via explicit synchronisation.

Under the topic of metacomputing several projects, like Harness [BDF+99], aim at providing functionality similar to *GRID-GUM2*. The characteristic difference to *GRID-GUM2* is the automatic management of parallelism within one parallel program.

Alt *et al* apply skeletons to computational Grids [ABG02]. This work focuses on providing the application user with skeletons to capture common patterns of Grid abstractions. However, *GRID-GUM2* provides more general programming language support for parallelism through an implementation that incorporates new implicit dynamic coordination-management strategies. Aldinucci *et al* also apply skeletons to computational Grids [ADD04]. This work focuses on providing a skeleton to centralise load management in the Grid environment. However, *GRID-GUM2* solves load scheduling on the Grid by developing a dynamic decentralised load schedule.

Chapter 3

*GRID-GUM*1: an Initial Grid Port of Parallel Haskell

3.1 Introduction

This chapter describes the design and implementation of adapting the Gum parallel runtime system for execution on the Globus Toolkit Grid middle-ware, *GRID-GUM*1. Gum and Globus Toolkit have been described in Chapter 2.4 and Section 2.1.4 respectively. We assess the performance of *GRID-GUM*1 and present some of the first systematic performance measurements of several high-level parallel programs on both homogeneous and heterogeneous computational Grids.

In earlier work [LRS+03], Loidl *et al* compared the performance of GpH program under Gum with C program. The benchmark program used in this comparison is `matMult`. The parallel C version of `matMult` is implemented in C+Pvm using the Gnu Multi-Precision library and the GNU C compiler. The comparison on 16 PEs shows better performance only of a factor of 1.6 for C+Pvm program relative to the GpH program. While the sequential performance of the C program is better by a factor of 5, the speedup values progress in a similar way as for the Gum program. In contrast, the parallel C+Pvm program size differs substantially. It is a factor of 6 longer than the GpH program.

This chapter is structured as follows: Section 3.2 presents *GRID-GUM*1. It discusses in Sub-Section 3.2.1 the GUM integration to Globus Toolkit GRID middle-ware and in Sub-Section 3.2.2 the working mechanism of *GRID-GUM*1. Section 3.3 evaluates the performance of *GRID-GUM*1. It compares in Sub-Section 3.3.2 *GRID-GUM*1 performance in a single cluster with GUM implementations on PVM and MPI and analyses *GRID-GUM*1 performance in computational GRID architectures, with low-latency communication in Sub-Section 3.3.3 and high-latency communication in Sub-Section 3.3.4. Section 3.4 concludes and discusses the behaviour and the weaknesses of *GRID-GUM*1.

3.2 *GRID-GUM*1

3.2.1 Integration

In developing *GRID-GUM*1, a crucial first step was to ensure that GUM could seamlessly support GPH in computational GRIDs. In particular, it was important to demonstrate conclusively that the high performance computer oriented GUM communication layer could be modified to be used in computational GRIDs. GUM's communication layer is based on PVM, however, the Globus Toolkit communication is based on a special form of MPI, MPICH-G2, discussed in Section 2.1.6. This leads to change the communication layer in GUM as a first step towards implementing *GRID-GUM*1. From a communication point of view, one can distinguish three different implementations of GUM: GUM/PVM, GUM/MPI and GUM/MPICH-G2 (*GRID-GUM*1):

GUM/PVM**:**

GUM/PVM presents the original implementation of GUM. It uses PVM for communication. As distinctive feature, GUM/PVM includes a *system manager* to control the startup and the termination of the execution. At the startup, the system manager spawns the PEs to start the execution. When the main thread

terminates, the mainPE indicates this by communicating with the system manager which then initiates and coordinates the shutting down of all PEs.

GUM/MPI:

GUM/MPI is an intermediate implementation between GUM/PVM and *GRID-GUM1*. It uses the MPICH implementation of MPI, for communication. GUM/MPI does not include a system manager. It uses MPI libraries to control the startup and the termination of the execution. At startup, GUM/MPI calls `mpi_init()` to spawn the PEs to start the execution. At the end of the execution when the main thread terminates, the mainPE calls `mpi_finish()` to shut down all PEs.

GUM/MPICH-G2 (*GRID-GUM1*):

GUM/MPICH-G2 is GUM implementation over the GRID, *GRID-GUM1*. *GRID-GUM1*'s communication layer is built around MPICH-G2, and hence the Globus Toolkit as middle-ware. Like GUM/MPI, *GRID-GUM1* does not comprise a system manager. It relies on MPICH-G2 to control the startup and the termination of execution. *GRID-GUM1* uses the same communication management as GUM/PVM and GUM/MPI and indeed all other managements are unchanged.

Figure 3.7 outlines how *GRID-GUM1* integrates with the GRID's layers, identified in Section 2.1.5. *GRID-GUM1* is located above all layers provided by the Globus Toolkit. It integrates with the GRID through the connectivity layer that provides a unified distributed environments comprising the underlying GRID. Such integration depends on the provision of MPICH-G2 to link *GRID-GUM1* transparently to the GRID. The next Sub-Section describes in more detail how *GRID-GUM1* works and interacts with the GRID layers.

3.2.2 *GRID-GUM1* Working Mechanism

Globus Toolkit requires a more elaborate machine configuration phase and this has to be reflected in *GRID-GUM1*. Prior to start parallel execution on *GRID-GUM1*, the user employs the GRID Security Infrastructure (GSI), described in

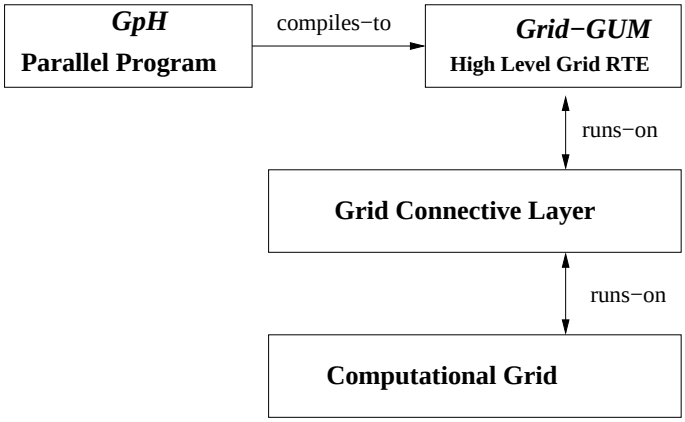


Figure 3.7: *GRID-GUM*1 system architecture

Section 2.1.4, to obtain a public-key proxy credential that is used to authenticate the user to each site. Once authenticated, the user issues the GUM execution command to start the parallel execution. After this, *GRID-GUM*1 uses the standard `mpirun` command in MPICH-G2 to request the creation of an MPI computation. The MPICH-G2 implementation of this command uses RSL scripts, identified in Section 2.1.4, to describe the job. The implementation of the *GRID-GUM*1 driver has been changed to use `mpirun` to construct internally RSL scripts. The RSL scripts identify resources (e.g., PE name, accessing port, certificate name), specify requirements, and parameters (e.g., location of executables, command line arguments, environment variables) for each. Then *GRID-GUM*1 passes the RSL script to the MPICH-G2 library to spawn the specified number of PEs. Based on the information found in the RSL script, MPICH-G2 calls a co-allocation library distributed with the Globus Toolkit, DUROC, illustrated in Section 2.1.4, to start the parallel execution across the PEs specified in the RSL script. Once the parallel execution has started *GRID-GUM*1 uses a Globus Toolkit service, *Global Access to Secondary Storage* (GASS), through MPICH-G2 to direct standard output and error (stdout, stderr) streams to the user's terminal. Figure 3.8 illustrates the *GRID-GUM*1's computational GRID architecture identifying the MPICH-G2 and Globus Toolkit components.

Finally, *GRID-GUM*1 is configured to perform parallel execution in which all PEs are on the same side of a firewall and where PEs are on opposite sides of a

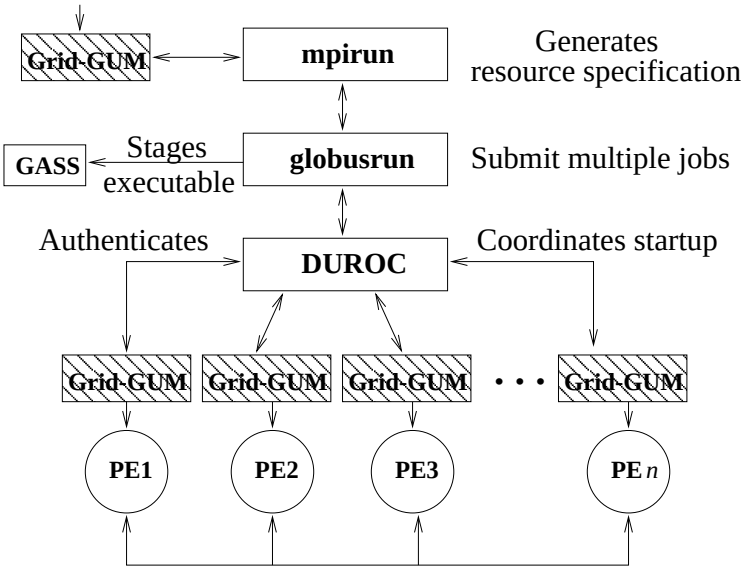


Figure 3.8: *GRID-GUM*1 components architecture

firewall. From a technical point of view, *GRID-GUM*1 uses the Globus Toolkit to address the main two issues that arise in the presence of firewalls which job control (e.g., start-up, and termination) and TCP messaging during execution. In a brief description, to run parallel execution over firewalls using *GRID-GUM*1, system administrators in each site have to open certain port numbers in the firewall, which called controllable ephemeral ports in the Globus Toolkit documents, and then *GRID-GUM*1 specify the port range has been opened with the environment variable `GLOBUS_TCP_PORT_RANGE` in the RSL scripts.

3.3 *GRID-GUM*1 Evaluation

3.3.1 Evaluation Framework

While there is some evidence that PVM and MPI offer comparable behaviours, it was not known whether the additional GRID layers might add unacceptable overhead costs to *GRID-GUM*1, rendering its use inappropriate for parallel functional programming.

The measurements are presented in this chapter have been performed on five Beowulfs clusters: three located at Heriot-Watt Riccarton campus (*Edin1, Edin2,*

and *Edin3*), a cluster located at Ludwig-Maximilians University Munich (*Muni*), and a cluster located at Heriot-Watt Borders campus(*SBC*); see Appendix A.1 for more details.

The programs measured in this chapter are classified by the communication degree, which is the number of messages the program sends per second, so we can study the impact of the latency of the communication on program behaviour. Six programs are measured in this experiment. Three have low-communication degree,

- `parFib`: the `parFib` program computes the number of calls to the Fibonacci function. The granularity of `parFib` can be controlled by specifying the threshold for parallel invocation which help manage the computation size. `parFib` is a divide-conquer program with regular parallelism.

- `queens`: the `queens` program places queen chess pieces on the chess board so that they do not check each other. `queens` is a divide-conquer program with regular parallelism.

- `sumEuler`: the `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. It is data parallel and has a fairly cheap combination phase involving only a small amount of communication. `sumEuler` has a high irregular parallelism.

and the other three have relatively high-communication degree,

- `raytracer`: the `raytracer` program calculates a 2D image of a scene of 3D objects by tracing all rays in a window. In tracing a ray, the intersections with the objects are computed. When an intersection is found, the ray is reflected and the colour of the intersection point is computed based on the strength of the ray and on the texture of the object's material. `raytracer` is data parallel program with high irregular parallelism. `raytracer` generates most of the parallelism at the beginning of the execution.

- `matMult`: the `matMult` program multiples two matrices. Given two square

matrices of arbitrary precision integers $A, B \in \mathbb{Z}^{n \times n}, n \in \mathbb{N}$ find their product, i.e. a matrix $C \in \mathbb{Z}^{n \times n}$ such that $C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$. `matMult` is a divide-conquer program with regular parallelism.

- `linSolv`: the `linSolv` program finds an exact solution of a linear system of equations of the form $Ax = b$ where $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^{n}, n \in \mathbb{N}$. `linSolv` is a symbolic algebra problem with data parallel paradigm and limited irregular parallelism.

The code of these programs can be obtained from [Loi01a].

All run-times in the coming tables represent the median of three executions to ameliorate the impact of operating system and shared network interaction.

3.3.2 Single Cluster

This experiment investigates the impact of using different communication libraries in the presence of additional GRID layer on the performance on a single cluster.

The measurements in this section have been performed on the Edin1 Beowulf cluster. In Table 3.2, the second column shows the different GUM implementations using PVM, MPICH and MPICH-G2 as communication library, respectively. The third column records the sequential runtime, and the fourth column records the parallel runtime on 16 PEs. The fifth and sixth columns record the wall-clock and execution speedup. The wall-clock time is the execution time plus the startup time. The seventh and the last columns show the percentage variance of the wall-clock and execution speedup relative to the GUM/PVM implementation speedup.

Table 3.3 summarises the dynamic properties of the programs presented in the experiments executed on a 16-processor Beowulf cluster. The second column shows the different GUM implementation, with different communication library. The third column records the total number of threads generated during the execution. The remaining columns show averages over all processors for the allocation rate, i.e. the amount of local memory allocated per second of execution time, the

Program Name	Comm Library	Runtime		Speedup		%Variance	
		Seq sec	16 PE sec	Wall Clock	Exec	Wall Clock	Exec
parFib	PVM	413.7	22.8	14.8	17.1		
	MPI	409.4	20.5	6.8	19.8	54%	-15%
	MPICH-G2	465.1	26.3	2.3	17.6	84%	-2%
sumEuler	PVM	1607.1	131.8	11.1	12.1		
	MPI	1585.1	139.2	8.8	11.3	20%	6%
	MPICH-G2	1598.1	188.1	3.5	8.4	68%	30%
raytracer	PVM	2855.4	315.3	8.9	9.6		
	MPI	2782.7	365.2	7.8	8.9	12%	7%
	MPICH-G2	2782.7	301.7	6.8	9.2	22%	4%
linSolv	PVM	834.2	102.6	6.5	8.9		
	MPI	828.4	110.5	5.5	7.3	15%	17%
	MPICH-G2	828.9	112.2	5.1	7.3	21%	17%
matMult	PVM	891.9	150.2	5.9	5.9		
	MPI	891.9	191.9	4.6	4.6	21%	21%
	MPICH-G2	916.3	292.6	3.1	5.0	47%	15%
queens	PVM	2802.7	375.1	7.4	7.4		
	MPI	2802.7	390.9	7.1	7.1	4%	4%
	MPICH-G2	2816.4	567.8	4.9	6.2	33%	16%

Table 3.2: Speedup on 16 PEs

communication degree, i.e. the number of packets sent per second of execution time, and the average packet size, i.e. the size of packet in Byte.

The behaviour of GPH programs with the different GUM implementations as shown in Table 3.2 consistently give the GUM/PVM implementation the best wall-clock speedup and *GRID-GUM*1 with MPICH-G2 using Globus Toolkit middleware the worst. As shown in measurements in the Table 3.3, the average packet size is relatively small for `parFib`, and `sumEuler`, and the wall-clock speedup variance is big between the different GUM implementations for these programs. For `raytracer`, `matMult`, and `linSolv` the average packet size is significantly larger and the wall-clock speedup variance is smaller.

The main source of overhead for the communication is the time needed for packing and unpacking in the communication libraries. Good performance for

Program Name	Comm Library	No of Threads	Alloc Rate MB/s	Comm Degree Msgs/s	Average Pkt Size Byte
parFib	PVM	26595	55.3	65.5	5.5
	MPI	26595	52.7	58.0	5.5
	MPICH-G2	26595	43.2	14.8	5.6
sumEuler	PVM	82	52.8	2.09	90.2
	MPI	82	47.9	1.4	90.3
	MPICH-G2	82	45.7	0.7	90.2
raytracer	PVM	350	60.0	46.7	321.7
	MPI	350	61.4	45.5	320.4
	MPICH-G2	350	49.5	62.9	323.0
linSolv	PVM	242	40.3	5.5	290.6
	MPI	242	40.8	3.1	300.1
	MPICH-G2	242	26.5	2.5	276.3
matMult	PVM	144	39.0	67.3	208.8
	MPI	144	40.1	52.2	213.3
	MPICH-G2	144	40.0	31.2	209.3
queens	PVM	24	38.8	0.2	851.8
	MPI	24	37.0	0.2	818.9
	MPICH-G2	24	34.0	0.1	846.1

Table 3.3: Dynamic Program Properties on 16 PEs

small packets is important for GUM, since parallel functional programs have massive amounts of fine grained parallelism including many small messages. This is untypical for general parallel applications, and MPI implementations are usually tuned for the common case of large packet sizes. However, the big difference between GUM implementation of PVM and MPI on one side and *GRID-GUM*1 with MPICH-G2 in the other side is related to the extra startup security checking overhead which Globus Toolkit middle-ware adds for MPICH-G2

A comparison of the execution-time speedup of the GUM implementations with the different GPH programs shows that no implementation is always better than the others. However, the differences in execution-time speedup are less marked than the differences on the wall-clock speedup.

Summary

- For programs with long execution time the performance of GUM is independent of the communication libraries (Table 3.2);
- *GRID-GUM*1 with MPICH-G2 has a high startup cost relative to GUM with PVM or MPI (Table 3.2).

3.3.3 Computational GRIDs: Low-Latency

	raytracer			queens(13)		
	Speedup		Runtime	Speedup		Runtime
	F	S	Sec.	F	S	Sec.
F	1.0	3.3	1483.3	1.0	3.2	719.5
S	0.3	1.0	4894.0	0.3	1.0	2324.7
FF	1.9	6.3	772.8	1.8	6.0	384.6
FS	1.2	4.0	1199.4	0.9	3.0	753.5
SS	0.5	1.8	2698.5	0.6	1.9	1176.9
SF	0.7	2.3	2106.1	1.0	3.4	666.3
FFF	2.7	8.9	545.1	2.8	9.3	249.5
FFS	2.0	6.7	728.6	0.9	3.0	768.2
FSS	1.4	4.8	1002.3	0.9	3.1	733.6
SSS	0.8	2.9	1663.0	0.9	2.9	795.7
SSF	1.4	4.6	1047.8	1.1	3.7	627.6
SFF	1.4	4.8	1002.1	1.5	4.8	478.3

Table 3.4: *a*) Heterogeneous low-latency Computational GRID

This experiment investigates the performance impact of executing GPH programs using *GRID-GUM*1 on heterogeneous computational GRIDs with moderate latency communication.

The measurements in Tables 3.4 and 3.5 use *GRID-GUM*1 on SBC and Edin3 Beowulf clusters. Each SBC machine is labelled S (Slow) and each Edin3 machine is labelled F (Fast). Two programs are measured: `raytracer` with relatively high-communication degree, and `queens` with relatively low-communication degree. The first column shows different combinations of machines. The second and

	raytracer			queens(13)		
	Speedup		Runtime	Speedup		Runtime
	F	S	Sec.	F	S	Sec.
FFFF	3.4	11.4	425.9	2.7	9.0	258.2
FFFS	2.7	9.0	538.8	1.4	4.8	483.0
FFSS	2.2	7.2	675.7	1.2	4.0	578.0
FSSS	1.7	5.8	833.1	1.2	4.1	561.6
SSSS	1.1	3.8	1280.9	0.9	3.1	741.6
SSSF	1.6	5.3	916.2	1.2	4.1	560.3
SSFF	1.4	4.8	1006.7	1.2	4.1	563.5
SFFF	1.4	4.6	1046.3	1.9	6.1	375.5
FFFFF	4.0	12.9	376.7	4.0	12.8	181.1
FFFFS	3.5	11.5	422.9	2.8	9.1	254.5
FFFSS	2.9	9.4	519.2	1.3	4.2	544.9
FFSSS	2.4	7.9	615.3	1.3	4.3	530.1
FSSSS	1.9	6.4	755.6	1.2	4.0	577.7
SSSSF	1.7	5.7	850.7	1.2	4.1	560.5
SSSFF	1.8	6.2	786.0	1.5	4.9	474.3
SSFFF	1.8	6.1	790.4	1.9	6.1	375.4
SFFFF	1.9	6.5	747.6	2.2	7.3	316.5

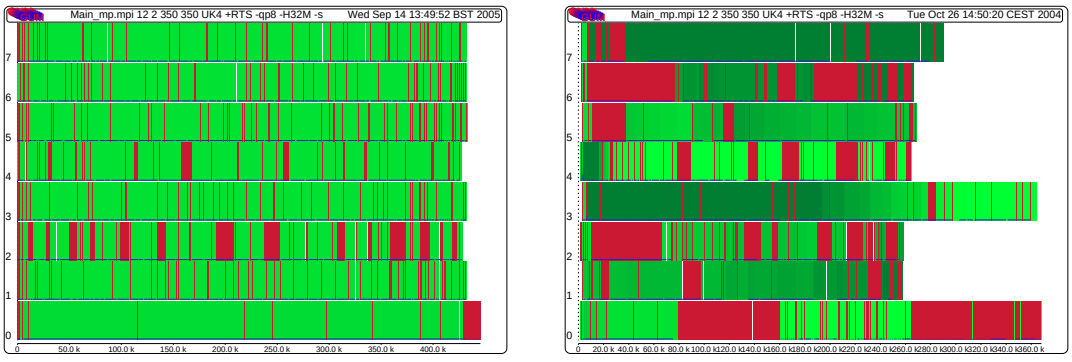
Table 3.5: *b*) Heterogeneous Low-Latency Computational GRID

the fifth columns record the speedup using F's sequential runtime for `raytracer` and `queens` respectively. The third and the sixth columns records the speedup using S's sequential runtime, and the fourth and the last columns show the wall-clock time. The first machine in the configuration string is where the program starts.

Tables 3.4 and 3.5 show that, replacing a local machine S by a faster remote machine F decreases the runtime and increases the speedup. For example in Tables 3.4 and 3.5, SSS machines requires 1663.0s to finish the computation of `raytracer`; however, if S machine has been replaced by F remote machine, the runtime is decreased by 37%. Interestingly, this result supports the idea of using a fast remote machine to improve the performance of a GPH parallel program, and it shows that *GRID-GUM*1 can cope with moderate latency communication without modification.

However, it is observable that *GRID-GUM*1, with its blind load distribution (see Section 2.4.4), often gives unsatisfactory distribution in heterogeneous computational GRIDs. For example, replacing one of the FFF machines by a slower remote machine S increases the runtime of `queens` from 249.5s to 768.2s, i.e. by a factor of three. Likewise, adding a slower remote machine S to two FF local machines increases the runtime of `queens` from 384.6s to 768.2s i.e. by a factor of two.

*GRID-GUM*1 shows relatively poor performance on heterogeneous computational GRIDs for many programs, and that is due to poor load distribution. Figures 3.9 and 3.10 show *GRID-GUM*1 per-PE and overall activity profiles for `raytracer` on homogeneous and heterogeneous computational GRIDs. A per-PE activity profile shows the behaviour for each of the PEs (y-axis) over execution time (x-axis) as described in Section B.2. An overall activity profile shows the behaviour of the program at each instant of its execution, as described in Section B.1. Figures 3.9.a and 3.10.a depict the performance on homogeneous computational GRIDs where all PEs have the same CPU speed. Figures 3.9.b and 3.10.b depict the performance on heterogeneous computational GRIDs where there are four fast machines (0-3) and four slow machines (4-7).

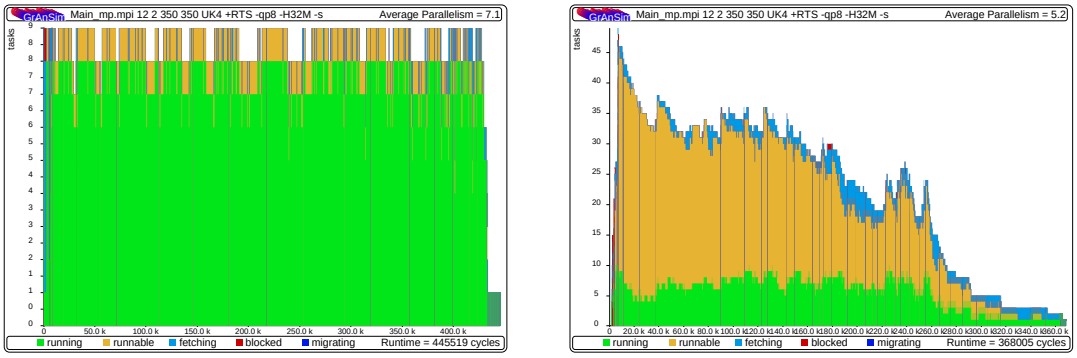


a) Homogeneous computational GRID *b*) Heterogeneous computational GRID

Figure 3.9: Per-PE Activity Profile for `raytracer`

All PEs in Figure 3.9.a are uniformly loaded, and finish at about the same

time, in contrast the PEs in Figure 3.9.b have numerous idle periods, and finish at different times. Figure 3.9.a also shows idle periods at the beginning of the computation, where only a small amount of parallelism is available, and blocking on data that is remotely evaluated will cause the entire PE to remain idle until new work is obtained (see the start of PE 6). Matching the profile in Figure 3.9.a, the fast processors in Figure 3.9.b (0-3) show a better balanced load and finish at about the same time. Towards the end only PE 3 has useful work, and the mainPE 0 has to wait for it to finish. Considering the runtime of the heterogeneous computational GRIDs, 368.0s, is almost two times greater than the runtime of the homogeneous computational GRIDs, 220.0s.



a) Homogeneous computational GRID *b*) Heterogeneous computational GRIDs

Figure 3.10: Overall Activity Profile for `raytracer`

Comparing the overall activity profiles in Figure 3.10.a and Figure 3.10.b shows that number of runnable threads in the heterogeneous computational GRIDs is far higher than the ones in the homogeneous computational GRIDs at each time which proves that the deterioration in the performance of the heterogeneous case is not related to the lack of parallelism in the GPH program. *GRID-GUM*1's blind load distribution allocates the same number of sparks to all PEs irrespective of CPU speed. This load distribution methodology lets PEs with slow CPU speed accumulate sparks in the spark pool and activate those sparks as that all

PEs involved in the execution have the same capability of execution and evaluating sparks. Note that, once a spark is been activated it can not be moved anywhere.

Summary:

- Replacing a local PE with a faster remote PE reduces execution time in all cases (Tables 3.4 and 3.5);
- *GRID-GUM*1's load distribution mechanism does not deliver good load balancing in a heterogeneous computational GRIDs (Figures 3.9 and 3.10);
- In a moderate latency configuration, latency is not the dominating factor, since *GRID-GUM*1 can overlap communication with computation, provided a sufficient amount of parallelism is available (Tables 3.4 and 3.5).
- The final data collection is one reason for the poor load distribution, as it causes the fast PEs to remain idle until the slow PEs finish their work (Figures 3.9 and 3.10).

3.3.4 Computational GRIDs: High-Latency

This experiment investigates the performance of executing GPH programs using *GRID-GUM*1 on homogeneous computational GRIDs with a high-latency communication. We measure programs with both low and high-communication degrees.

The measurements in Table 3.6, and 3.7 use *GRID-GUM*1 on the Muni and Edin2 Beowulf clusters. Each Muni machine is labelled M and each Edin2 machine is labelled E

Five programs have been tested: two programs with relatively low-communication degree `parFib`, and `sumEuler`, and three programs with relatively high-communication degree `raytracer`, `linSolv`, and `matMult`.

For programs with a low-communication degree, Table 3.6 shows that adding a remote machine M decreases the runtime. Even on computational GRIDs over

	parFib (45)			sumEuler		
	Runtime	Speedup		Runtime	Speedup	
	Sec.	E	M	Sec.	E	M
M	867.5	1.2	1.0	3138.5	1.0	1.0
E	1070.1	1.0	0.8	3227.6	1.0	0.9
MM	431.0	2.2	2.0	1270.4	2.5	2.4
EM	480.6	2.2	1.8	1308.8	2.4	2.3
EE	536.8	1.9	1.6	1332.8	2.4	2.3
MMM	298.8	3.5	2.9	869.9	3.7	3.6
EMM	331.1	3.2	2.6	838.7	3.8	3.7
EEM	338.9	3.1	2.5	867.9	3.7	3.6
EEE	374.8	2.8	2.3	899.7	3.5	3.4
MMMM	241.9	4.4	3.5	629.7	5.1	4.9
EMMM	251.3	4.2	3.4	670.7	4.8	4.6
EEMM	268.0	3.9	3.2	665.8	4.8	4.7
EEEM	274.9	3.8	3.1	665.5	4.8	4.7
EEEE	292.9	3.6	2.9	662.2	4.8	4.7
MMMMM	205.9	5.0	4.2	523.2	6.1	5.9
EMMMM	212.7	5.0	4.0	544.0	5.9	5.7
EEMMM	226.2	4.7	3.8	553.7	5.8	5.6
EEEMM	224.7	4.7	3.8	620.8	5.1	5.0
EEEEM	234.0	4.5	3.7	588.4	5.4	5.3
EEEEE	251.3	4.2	3.4	570.8	5.6	5.4

Table 3.6: Low-Communication Degree Programs

relatively high-latency communications, the additional computational power outweighs the expensive but infrequent communication. It also shows that replacing a local machine E by a remote machine M, yielding a EEM configuration, shows little change in the runtime (3.5%). Furthermore, replacing a local machine E by a remote machine M does not grossly deteriorate performance. For example, in Table 3.6, in an EEE configuration `sumEuler` requires 899.7s to finish, machine M is added $EEEM$ the runtime decreases by 26.0%. In short, using remote machines in high-latency communications does not have impact on the performance of low-communication degree programs.

Table 3.7 measures programs with a high-communication degree. Replacing a local machine with a slightly faster remote machine increases the runtime and

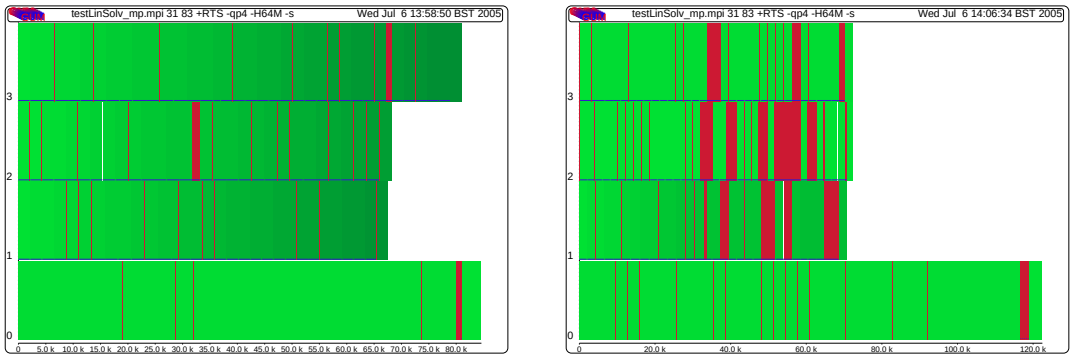
	raytracer			matMult			linSolv		
	Runtime	Speedup		Runtime	Speedup		Runtime	Speedup	
	Sec.	E	M	Sec.	E	M	Sec.	E	M
M	903.8	1.1	1.0	265.8	0.9	1.0	290.0	1.0	1.0
E	1027.8	1.0	0.8	259.9	1.0	1.0	299.3	1.0	0.9
MM	548.6	1.8	1.4	228.8	1.1	1.1	196.5	1.5	1.4
EM	624.6	1.6	1.4	393.0	0.6	0.6	232.0	1.2	1.2
EE	545.7	1.8	1.6	227.8	1.1	1.1	164.9	1.8	1.7
MMM	383.7	2.6	2.3	133.0	1.9	1.9	139.4	2.1	2.0
EMM	535.8	1.9	1.6	297.7	0.8	0.8	231.8	1.2	1.2
EEM	494.9	2.0	1.8	201.9	1.2	1.3	141.1	2.1	2.0
EEE	387.5	2.6	2.3	137.8	1.8	1.9	136.8	2.1	2.1
MMMM	312.8	3.2	2.8	121.9	2.1	2.1	119.5	2.5	2.4
EMMM	497.6	2.0	1.8	295.0	0.8	0.9	142.5	2.1	2.0
EEMM	421.7	2.4	2.1	213.9	1.2	1.2	134.9	2.2	2.1
EEEM	377.9	2.7	2.3	145.9	1.7	1.8	120.9	2.4	2.3
EEEE	326.4	3.1	2.7	114.8	2.2	2.3	117.1	2.5	2.4
MMMMM	287.8	3.5	3.1	108.6	2.3	2.4	104.4	2.8	2.7
EMMMM	473.8	2.1	1.9	290.8	0.8	0.9	147.0	2.0	1.9
EEMMM	413.7	2.4	2.1	228.8	1.1	1.1	142.1	2.1	2.0
EEEMM	378.7	2.7	2.3	150.9	1.7	1.7	104.9	2.8	2.7
EEEEM	329.9	3.1	2.7	125.1	2.0	2.1	107.7	2.7	2.7
EEEEE	279.8	3.6	3.2	95.9	2.7	2.7	102.9	2.9	2.8

Table 3.7: High-Communication Degree Programs

decreases the speedup. For instance `linSolv` on two local machines EE takes 174.9s, but if one of the local machines is replaced by a remote machine EM, the runtime increases by 41.4%. Note that for all programs the runtime increases when adding a remote machine in such a way. Furthermore, a configuration of the form $EMM\ldots M$ is always worst among the configurations with the same number of PEs. This is because the local machine E, which has all the work in the beginning of the execution, has to communicate with the other machines through a high-latency communication, which becomes a bottleneck in the execution. Finally, configurations of the form $E\ldots E$ or $M\ldots M$ are usually the best configurations, because all machines communicate with others through the low-latency communication. Clearly, to minimise latency, a local communication

is preferable.

*GRID-GUM*1 shows relatively poor performance on high-latency computational GRIDs for many programs. For example, Figures 3.11 and 3.12 show *GRID-GUM*1 per-PE and overall activity profiles for `linSolv` on a homogeneous low-latency and a heterogeneous high-latency computational GRIDs. Figures 3.11.a and 3.12.a depict the performance on homogeneous low-latency computational GRIDs. Figures 3.11.b and 3.12.b depict the performance on heterogeneous high-latency computational GRIDs where PE 0 & PE 1 and PE 2 & PE 3 are connected pairwise by a low-latency communication, and with a high-latency communication between the pairs.

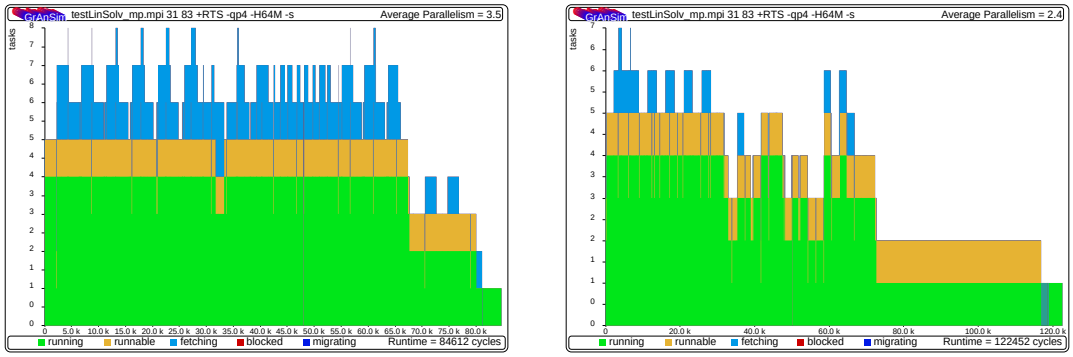


a) Low-Latency Communication *b*) High-Latency Communication

Figure 3.11: Per-PE Activity Profile for `linSolv` in Homogeneous Computational GRIDs

In Figure 3.11.b the PEs exhibit significantly more idle time, i.e. red colour in the horizontal line, and complete at different times. In contrast, the work is fairly evenly balanced in Figure 3.11.a. The idle time in Figure 3.11.b is due to PEs waiting for data to without having other threads execute.

Matching the overall activity profiles in Figure 3.12.b and Figure 3.12.a show the impact of the high-latency communication between PEs on the available running and runnable threads. In the high-latency setup, Figure 3.12.b shows bottlenecks from about 35k cycles. This is due to the *GRID-GUM*1's load distribution mechanism in high-latency setup. *GRID-GUM*1's load distribution mechanism



a) Low-Latency Communication *b*) High-Latency Communication

Figure 3.12: Overall-Activity Profile for `linSolv` in Homogeneous Computational GRIDs

does not have any monitoring technique to control the messaging process. Messages in *GRID-GUM*1 flow between PEs regardless of the latency between them. This results that PEs stay idle for fairly long time due to the reason that not enough work had been accumulated and that means poor load distribution.

Summary

- For high-communication degree programs *GRID-GUM*1 delivers poor performance on high-latency computational GRIDs (Table 3.7);
- For low-communication degree programs *GRID-GUM*1 can deliver good performance on high-latency computational GRIDs (Table 3.6);
- The poor performance of *GRID-GUM*1 on high-latency computational GRIDs is primarily due to poor load distribution (Figures 3.11 and 3.12).

3.4 Conclusion

This chapter includes systematic measurements of the performance of GpH, one of the few Grid-enabled high-level parallel languages, on a range of computational Grids. This chapter concludes that the *GRID-GUM*1 dynamic management performs reasonably well even on moderate latency computational Grids. Thus, even without elaborate mechanisms to track information about load and latency, the memory management overhead does not become prohibitively expensive, and a model of semi-explicit parallelism can be used to exploit computational Grids. However, for programs with high-communication degree several shortcomings of *GRID-GUM*1 are exposed.

More specifically, Tables 3.2 and 3.3 show that for programs with big execution-time, the performance of Gum on a single cluster is largely independent of the communication library used. While the overall differences in performance are small, our measurements showed that Gum/Pvm gives the best speedups, being more efficient in handling many small messages, and *GRID-GUM*1 with Mpich-G2 and using Globus Toolkit middle-ware performs worst, paying a higher overhead due to the reliance on Globus Toolkit and its deeper protocol hierarchy. Despite being designed for homogeneous computational Grids, *GRID-GUM*1 delivers good and predictable speedups, on computational Grids with a low-latency communication. In contrast, on computational Grids with a high-latency communication, *GRID-GUM*1 only delivers acceptable speedups for low-communication degree programs like `queens` (110.0 Byte/s), and little speedup for high-communication degree programs like `raytracer` (30326.39 Byte/s).

In comparing the runtime behaviour on various configurations, we identified two main problems. Several programs finish with an inherently sequential data collection phase, which is stretched significantly on systems with higher latency. Depending on the program, this can become a dominating factor and might motivate restructuring of the code. However, this is a program-dependent issue, and we cannot hope to solve this problem purely in the runtime environment, using a model of semi-explicit parallelism. A second problem is the poor load balance,

as discussed in Sections 3.3.3 and 3.3.4. GRID-GUM1's load distribution mechanism has been designed to work in a flat architecture with uniform PE speed and communication latency. In contrast, computational GRIDs are hierarchical, heterogeneous and shared. Therefore, making the runtime environment aware of the basic characteristics of the GRID-architecture, such as the speed of and the latencies between machines, can be used to improve the load balancing policy in system. Furthermore, these changes will be beneficial for all parallel programs, since they are integrated on system- rather than program-level.

In the next chapter we propose a model where the PEs maintain dynamically latency and load information to inform load management, so that work is only sought from PEs which are known to be relatively heavily loaded, giving preference to local cluster resources. To propagate the necessary information, we will augment the messages in GRID-GUM2 to carry dynamic information about latency and load between PEs, and hence between clusters. Such information will be combined with static PE characteristics to determine relative loads.

Chapter 4

Design of An Adaptive RTE for Computational GRIDs

4.1 Introduction

In this chapter, *GRID-GUM2*, an adaptive runtime environment for computational GRID architecture is presented. To the best of our knowledge, *GRID-GUM2* is the first fully implemented virtual shared memory runtime environment on computational GRIDs. We demonstrate that virtual shared memory is feasible on computational GRIDs and that it can deliver good speedups if combined with an aggressive dynamic load distribution mechanism. We present measurements that quantify the improvements on small, but realistic, computational GRIDs. The rest of this chapter discusses shortcomings of *GRID-GUM1* in Section 4.2, presents *GRID-GUM2* in Section 4.3, adds a monitoring mechanism to *GRID-GUM2* in Section 4.4, and finally develops an adaptive load distribution of *GRID-GUM2* in Section 4.5.

4.2 Shortcomings of *GRID-GUM1*

In Chapter 3 we investigated the performance of GPH programs on *GRID-GUM1*, with varying numbers of clusters and interconnection latencies in real

computational GRIDs. We found that *GRID-GUM*1 delivers good and predictable speedups for several configurations, in particular multiple homogeneous clusters with a low-latency communication, but also on multiple clusters with a high-latency communication for programs that perform little communication. However on other configurations, including heterogeneous clusters, *GRID-GUM*1 gives poor performance due to poor load distribution (Sections 3.3.3 and 3.3.4).

*GRID-GUM*1 is a system of passive load distribution, which means a PE can request work only if it is idle. It uses a "fishing" mechanism, in which requests for work are sent to random PEs. The advantage of this load distribution mechanism is its simplicity. No load information needs to be calculated, sent, received or stored. No heuristic functions are required to select which PE will receive the FISH message. All that is required is a random number generator and knowledge of the number of PEs participating in the computation.

The following disadvantages of *GRID-GUM*1's load distribution are exacerbated in computational GRIDs.

i. Firstly (*eager fishing*), sending a FISH message occurs every-time a PE is idle regardless of the PE's capabilities (CPU speed in this case). In heterogeneous computational GRIDs, if all PEs send FISH messages without monitoring control, the network is flooded with FISH messages. Valuable work might be stolen by weak PEs, leaving strong PEs idle. Additionally, the FISH messages can increase overall latency by saturating the network with messages. Such an increase in latency increases not only the time to find work, but also the time to send data to a PE blocked on another PE.

ii. Secondly (*dumb fishing*), the destination of the FISH message is random. There is, therefore no guarantee that the FISH message will be sent to a PE with work to donate, in preference of a PE with no work. In fact there is no guarantee that the message will reach an given PE at all. In computational GRIDs danger of missing PEs with work is typically, computational GRIDs contains a large number of PEs and an only slightly larger number of units of work. This shows that GUM was designed for a high ratio of

work/PEs. Thus, in *GRID-GUM1* a FISH message could bounce back and forth between a small group of PEs which do not have work to donate.

iii. Thirdly (*remote fishing*), the random destination of the FISH message might bring work from remote PEs through high-latency communication. Obtaining work from a PE with high-latency communication increases idle time. When searching for work, preference should be given to local PEs.

iv. Fourthly (*fixed-size fishing*), the recipient PE of the FISH message has no flexibility in deciding the amount of work that should be donated to idle PE. The fishing PE might be a local PE or belong to a remote cluster which can be a powerful cluster or a weak cluster according to the number of PEs. The amount of work returned to the fishing PE should not be the same in all cases. The recipient PE has to study the load and the capability of the fishing PE and then decide how much work should be send back.

v. Lastly (*blind-start fishing*), the blind way of nominating the main PE which controls the initialisation and termination of the program and the start of the execution of the main thread (see Section 2.4.2) in *GRID-GUM1*. The main PE is always chosen as the first PE in the PE list. All PEs at the beginning of the execution seek work by sending FISH messages to the main PE, in the extreme scenario, the main PE has relatively slow CPU speed and is located in a remote cluster with few PEs. That means most of the PEs have to obtain work from the main PE through high-latency communication, which has negative impact on the performance.

4.3 *GRID-GUM2*

Based on the results in Chapter 3 and the discussions in Section 4.2, it is essential to modify *GRID-GUM1* for execution on computational GRIDs, (*GRID-GUM2*). *GRID-GUM2* is a virtual shared memory runtime environment built over computational GRIDs. It is designed to monitor loads, latencies and physical characteristics like CPU speed in computational GRIDs. *GRID-GUM2* uses the monitored

information to provide a good load distribution over computational GRIDs using the following policies:

- An idle PE sends a FISH message only to a PE that has high load relative to its CPU speed.
- PEs have a preference for obtaining work from PEs that currently have low-latency communication.
- The recipient PE switches from passive to active load distribution if a FISH message is received from another cluster.
- *GRID-GUM2* starts the computation in the biggest cluster which has more PEs with fast CPU speeds than the other clusters.

GRID-GUM2 is designed to work in a closed computational GRID, which means it is not possible for other machines to join the computation after it has started. Also it is tuned for a setup typically found in high-performance clusters. It is designed to work most effectively in: *a)* non-shared machines where only one program can execute at a time, *b)* dedicated machines which means the machines are fully booked for a specific program, and finally *c)* non-preemptive, which means once the program started on such resources, it can not be preempted until it completes the execution.

The *GRID-GUM2* mechanism has two main components: information collection and adaptive load distribution. Information collection is supported by a monitoring mechanism to provide the current state information of the computational GRID. The monitoring mechanism operates during the whole course of execution. It collects static information like CPU speed and IP addresses at program start up, and dynamic information such as load and latency during execution. Adaptive load distribution of *GRID-GUM2* comprises the following aspects:

- Resource-level load distribution: programs executed in *GRID-GUM2* do not required specific resource. Idle PEs then use load distribution mechanism to seek work from PEs relatively heavily loaded.

- Dependent load distribution: *GRID-GUM2* aims for an efficient load distribution mechanism to a single parallel program with dependent tasks.

- Decentralised information services: *GRID-GUM2* maintains a decentralised scheme where every PE is responsible for maintaining state information of some nearby PEs and sharing it with other PEs.

- Dynamic load distribution: *GRID-GUM2* assumes that limited knowledge about the load and PEs are available *a priori*, and load distribution decisions have to be made during the execution.

- Decentralised load distribution organisation: *GRID-GUM2* distributes the load distribution decision to every PE. Therefore, each PE acts as both a load distributer and a computational resource.

- Redistribution support: *GRID-GUM2* supports work placement which enhance system reliability and flexibility.

- Adaptive load distribution: *GRID-GUM2* is a mainly passive load distribution system where lightly loaded PEs have to explicitly ask for work from PEs with excess load. However, if an idle PE requests work from a PE residing outside its cluster and the request originated from a relatively powerful cluster, *GRID-GUM2* changes from a passive to an active system and the recipient PE sends more work to the idle PE.

The *GRID-GUM2* implementation uses the original implementation of GUM memory management based on a virtual shared heap, described in Section 2.4.5. The communication libraries in *GRID-GUM2* are built around MPICH-G2 and hence the Globus Toolkit as middle-ware, these are inherited from *GRID-GUM1*. *GRID-GUM2* presents a new load distribution to improve the performance on computational GRIDs. The load distribution is supported by a new monitoring mechanism.

The salient features of the monitoring mechanism and the load distribution of *GRID-GUM2* are explained in the following sections.

4.4 Monitoring Mechanism of *GRID-GUM2*

The monitoring mechanism used by *GRID-GUM2* is designed to be simple and effective in a computational Grid architecture. The information monitored by *GRID-GUM2* comprises of two types of information:

- static load information which includes CPU speed, the number of clusters and the number of PEs in each cluster, and
- dynamic load information which includes the number of sparks in each PE (load) and the latency between PEs.

This information is used by the new adaptive load distribution to improve the performance of *GRID-GUM2*. The following sections describe how the static and dynamic information are collected and made available on all the PEs and how the new load distribution profits from these information.

4.4.1 Static Load Information

Each PE maintains static load information about PEs and clusters that participating in the computation in a local table *PEStatic*. The information in PEStatic is collected only once at the beginning of the execution. Figure 4.8 shows a sketch of PEStatic table for the PEs in Figure 4.13

At the beginning of the execution, each PE collects and registers its PEId, CPU speed and IP address in the PEStatic table. PEId is a unique number generated by the communication library, Mpich-G2, described in Section 2.1.6, to distinguish between PEs in the case of sending and receiving messages. CPU speed and IP address are collected from the local system files `"/proc/cpuInfo"` and `"/etc/hosts"` respectively. After collecting the CPU speed and IP address, each PE broadcasts its current available static information (PEId, CPU speed and IP address) to the other PEs in the computational Grids.

PEStatic table stores a starting clock time for each PE to help in estimating the latencies between PEs as described later in this section. Hence, the starting

clock time is considered to be different between PEs in different geographical location, also some PEs have different setup time than others within the same cluster. PEs synchronises their starting clock time at the moment when all PEs are at the same execution level. In more details, the starting clock time is synchronised when PEId, CPU speed and IP address have been broadcast to all PEs. Each PE stores its starting clock time locally in the PEStatic table under *startClockTime*, and then broadcasts it to the other PEs.

The adaptive load distribution of *GRID-GUM2* clusters PEs statically according to the IP address, and nominates a main PE to control the starting and the termination of the execution. Section 4.5 explains how *GRID-GUM2* clusters PEs and chooses the main PE. The cluster Id and the nominee main PE are stored in PEStatic table under *clusterId* and *theMainPE* respectively. Table 4.8 sketches PEStatic Table for the computational GRIDs presented in Figure 4.13.

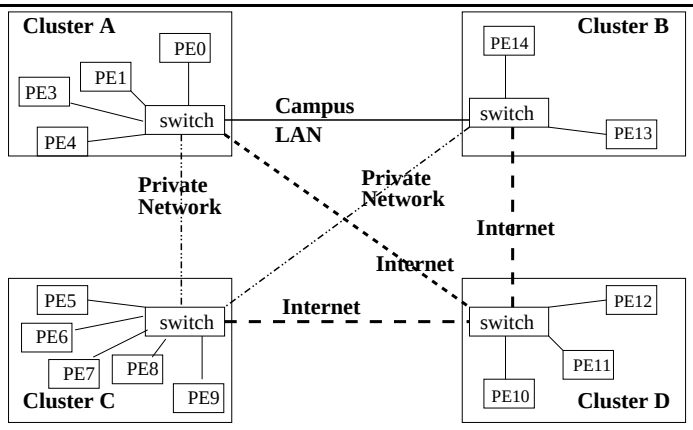


Figure 4.13: Computational GRIDs

The PEStatic table in each PE carries a full static information about all PEs in the computational GRIDs before the start of the execution of the main thread. The adaptive load distribution in *GRID-GUM2* uses these static information to improve the performance of a parallel program in the computational GRIDs as described in section 4.5.

PE-Id	cluster-Id	PE-CPU-Speed	cluster-CPU-Speed	PE-IP-Address	the-Main-PE	start-Clock-Time
0	1	568.2	3343.6	xxx.xxx.149.456	5	12:01:05.01
1	1	560.4	3343.6	xxx.xxx.149.456	5	12:01:05.11
2	1	569.8	3343.6	xxx.xxx.149.456	5	12:01:05.01
3	1	561.8	3343.6	xxx.xxx.149.456	5	12:01:05.20
4	1	565.5	3343.6	xxx.xxx.149.456	5	12:01:05.50
5	2	2569.9	12834.1	xxx.xxx.256.456	5	11:01:04.45
6	2	2563.4	12834.1	xxx.xxx.256.456	5	11:01:04.11
7	2	2565.6	12834.1	xxx.xxx.256.456	5	11:01:04.81
8	2	2566.2	12834.1	xxx.xxx.256.456	5	11:01:04.76
9	2	2569.0	12834.1	xxx.xxx.256.456	5	11:01:04.80
10	3	1802.1	5407.0	xxx.xxx.566.149	5	13:01:15.41
11	3	1800.9	5407.0	xxx.xxx.566.149	5	13:01:15.50
12	3	1804.0	5407.0	xxx.xxx.566.149	5	13:01:15.23
13	1	258.2	3343.6	xxx.xxx.149.456	5	12:01:05.44
14	1	259.7	3343.6	xxx.xxx.149.456	5	12:01:05.28

Table 4.8: A sketch of PEStatic Table for the PEs in Figure 4.13

4.4.2 Dynamic Load Information

Dynamic information is maintained in two tables, *PEDynamic* and *ComMap*. PE-Dynamic and ComMap contain partial dynamic information about processes and communication respectively. This information is exchanged in FISH messages.

PEDynamic: Load Information

Figure 4.14 sketches the PEDynamic table. It lists load information of all PEs. As discussed in Section 4.3, *GRID-GUM2* is designed to work most effectively in a dedicated system which means the machines are fully booked for a specific program. Thus, the PEDynamic table collects the load of the executing program, instead of monitoring the external load. Moreover, monitoring the entire load of a PE is not a sophisticated task. Many languages like C have built-in libraries which monitor external load. Due to this, the PE load is presented by the number of sparks in the spark pool. This is indicated in the PEDynamic table by the *noOfSparks*. As explained in Section 2.4.3, the spark represents a potential

parallelism.

PE	noOfSparks	timeStamp
1	2000	14:13:49
4	3000	14:13:59
3	10000	14:12:22
⋮	⋮	⋮

Figure 4.14: PEDynamic Table

The load information in PEDynamic must be maintained dynamically, but it is prohibitively expensive to regularly broadcast complete load information to every PE in large computational GRIDs. The monitoring mechanism of *GRID-GUM2* uses a lightweight approach maintaining partial dynamic load information at almost no cost. Each PE maintains its load information in the PEDynamic table and a time stamp (*timeStamp*) to avoid replacing recent load information with old. To circulate the load information over the computational GRIDs the PEDynamic table is included with every message sent between PEs. The receiving PE of the PEDynamic table updates its table with more recent information by comparing the timeStamp record in the receiving PEDynamic table with the local one. The additional cost of sending the PEDynamic table is minimal for the high bandwidth interconnects in a typical computational GRIDs. At the beginning of the execution the PEDynamic table lacks load information about PEs. However shortly after the start of the execution, the exchanging of messages between PEs gradually increase the level of load information in the PEDynamic in each PE.

ComMap: Communication Latency

In a computational GRID with PEs distributed over a wide area network, latency increases and gets less predictable. Drum *et al* [DR00] conclude that, along with the improvement of global communication infrastructures, the available bandwidth increases, however, this is not true for latency time (in fact the latency time might decrease but not in the same magnitude of bandwidth and computing

power increase). Hence, the monitoring mechanism of *GRID-GUM2* monitors only *latency* between PEs in the ComMap table. Figure 4.15 sketches ComMap table.

PE	Latency
1	0.75 msec
3	2.05 msec
6	10.00 msec
⋮	⋮

Figure 4.15: ComMap Table

Each PE collects in its local ComMap table the latency information from every PE in the computational GRIDs dynamically and in a lazy fashion during the course of the execution. The latency is calculated by measuring message send and receive times as time elapsed. Every time a PE sends a message, it attaches the message send time to the message. The recipient PE records the the message receive time to calculate the latency.

Each of the send and receive PEs use its local clock to record the message send and receive time. In computational GRIDs the local clock of the PEs are not synchronised. To overcome this problem, the recipient PE synchronises the message send and receive times using the start clock time which recorded in the PEStatic table as identified before in Section 4.4.1. The recipient PE calculates after how long the message is been generated according to the execution time of the program by subtracting the sending PE start clock time from the message send time. After that the recipient PE calculates the message send time according to its local clock time by adding the result of the last subtraction to its start clock time. Finally, the recipient PE calculates the latency by subtracting the new message send time from the message receive time.

The ComMap table profits from the broadcast of the static information at the beginning of the program to calculate initial values of latencies. The latency values in the ComMap Table are sorted directly after any update. The idea is to

keep PEs with low-latency values on the top of the table. Due to the sorting in the ComMap Table, PEs are ranked according to their priority for communication when there is a need to seek work from another PE.

4.5 Adaptive Load Distribution of *GRID-GUM2*

The adaptive load distribution of *GRID-GUM2* deals with:

- startup,
- work locating, and
- work request handling mechanisms.

Each of these mechanisms now will be discussed.

4.5.1 Startup Mechanism

The startup mechanism in *GRID-GUM1*, as shown in Figure 4.16, is based on nominating always PE0 as the main PE to start the execution of the main thread. In computational GRIDs, PE0 might have slow CPU speed and be located in a remote cluster with few PEs. That means, as explained in Section 4.2, most of the PEs have to obtain work from PE0 through high-latency communication, which has a negative impact on performance.

```
mainPE = PE0
IF mainPE THEN
  start computation
ELSE
  send FISH message to mainPE
```

Figure 4.16: *GRID-GUM1* Startup algorithm

In contrast, the adaptive load distribution in *GRID-GUM2* nominates the main PE, by finding the PE with the fastest CPU speed within the cluster which has the biggest clusterCpuSpeeds in the PEStatic table. Figure 4.17 illustrates

the startup mechanism in *GRID-GUM2* in pseudocode, the bold font is used to show the new adaptive load distribution.

```
collect static Info in PEStatic Table
broadcast static Info to all PEs
calculate and store latencies to all PE in ascending order in
    ComMap Table
cluster PEs statically
find mainPE
broadcast clusterId and mainPE to all PEs
IF mainPE THEN
  collect load Info in PEDynamic Table
  start computation
ELSE
  initial load Info = 0
    in PEDynamic Table
  send FISH message to mainPE
```

Figure 4.17: *GRID-GUM2* Startup Algorithm

The startup mechanism in *GRID-GUM2* clusters PEs statically according to the IP address. PEs that share the same last two subnets are classified as they are in the same cluster and have the same *clusterId* in the PEStatic table. For instance as depicted in Figure 4.13 and Table 4.8 PEs in cluster A and Cluster B have been classified under the same cluster. After that, *GRID-GUM2* calculates the total CPU speed for each cluster and stores it under *clusterCpuSpeeds* in the PEStatic table, see Table 4.8 for example. At this moment it selects the nominee main PE according to the criteria explained before. As show in Table 4.8 the main PE is the PE with PEId 5, PE5, which is the fastest PE in cluster 2 which has the biggest clusterCpuSpeeds of all clusters. The nominee main PE is stored in PEStatic table under *theMainPE*. Finally before theMainPE starts the execution, it broadcasts clusterId, clusterCpuSpeeds and theMainPE to all PEs in the computational GRIDs.

4.5.2 Work Location Mechanism

The work location mechanism in *GRID-GUM*1 identifies randomly a destination PE to donate work, as shown in Figure 4.18. *GRID-GUM*1 adopts a crude policy of work location by forcing all PEs to choose PE0 (the main PE) as a destination of the FISH message until PE0 fails to donate any work. At that moment PEs choose randomly a destination PE for their FISH messages.

```
IF idle (localPE) THEN
  IF runnable thread THEN
    evaluate new thread
  ELSE
    IF spark in spark pool THEN
      activate new spark
    FI
  FI
  IF noOfSpark < low-watermark THEN
    IF last SCHEDULE from mainPE THEN
      destPE = mainPE
    ELSE
      destPE = random PE from PEs list
    FI
    send FISH to destPE
  FI
FI
```

Figure 4.18: *GRID-GUM*1 Work location Algorithm

The *GRID-GUM*1 policy of work location is not scalable especially in computational GRIDs with increasing number of PEs. This policy raises the disadvantages explained in Section 4.2.

However, the adaptive load distribution of *GRID-GUM*2 follows a different policy for work location mechanism. It uses static information about CPU speeds and dynamic information about loads and latencies to source work from PEs that have high load relative to their CPU speeds and with preference for a PE with low-latency communication. Figure 4.19 illustrates the work location mechanism in *GRID-GUM*2 in pseudocode level, the bold font is used to show the new adaptive load distribution.

```
IF idle THEN
  IF runnable thread THEN
    evaluate new thread
  ELSE
    IF spark in spark pool THEN
      activate new spark
    FI
  IF noOfSpark < low-watermark THEN
    update local load Info in PEDynamic Table
    calculate localRatio = (localSpeed/localLoad)
    sort latencies in ascending order in ComMap Table
    FOR each PE in comMap Table
      calculate destRatio = (destSpeed/destLoad)
      IF localRatio > destRatio THEN
        attach to FISH: messageSendTime, PEDynamic Table
      FI
       send FISH to destPE
        break
    end FOR
    IF NOT mainPE AND NO FISH message sent THEN
     destPE = mainPE
      attach to FISH: messageSendTime, PEDynamic Table
     send FISH to destPE
    FI
  FI
FI
```

Figure 4.19: *GRID-GUM*2 Work location Algorithm

In *GRID-GUM*1 and *GRID-GUM*2, if a PE is idle, then firstly it should check if there is any thread waiting to be evaluated in the thread pool. If there are no threads to be evaluated, it checks if there is any spark can be activated in the spark pool. In the case when there are no sparks to be activated or the number of sparks in the spark pool are below the low-water mark, the PE has to seek work from another PE. In *GRID-GUM*1, the PE sends a FISH message randomly to another PE seeking work. However, in *GRID-GUM*2 the PE has to find a proper destination PE according to the available static and dynamic information in PEStatic and PEDynamic tables respectively.

In *GRID-GUM*2, according to Figure 4.13 and Table 4.8, if `PE0` in `cluster`

`A` needs to send a FISH message it updates its local PEDynamic table with its current load. Then it calculates its *localRatio*, which is the relative load (noOfSparks) to the CPU speed, from PEDynamic and PEStatic tables respectively. The records in the ComMap table are sorted in ascending order according to the latency values. After that, `PE0` selects the first PE in the ComMap table, `PE1` in `cluster A`, which should have the lowest latency communication with `PE0`. Before `PE0` sends FISH message to `PE1`, it examines `PE1` capability of donating work to `PE0`, by calculating *destRatio* of `PE1`, which is the ratio between the CPU speed and loads from PEStatic and PEDynamic tables respectively. To approve sending the FISH message to seek work to `PE1` from `PE0` is that `PE0`'s localRatio should be greater than `PE1`'s destRatio. If `PE1` appears to be incapable to donate work to `PE0`, `PE0` goes back and selects the next PE in the ComMap table. In the case when all PEs in the ComMap table prove to be incapable to donate work to `PE0` according to the CPU speeds and the loads information, `PE0` sends FISH message to the main PE (`PE5`). The reason that `PE0` sends the FISH message to the main PE as a final solution for searching of a destination PE are:

- it might be the beginning of the execution and the only PE that has work at this stage is the main PE,
- the main PE might have more precise information about loads in other PEs due to its centralisation role at the beginning of the execution.

In *GRID-GUM2*, a FISH message before is sent, it has to be tagged with the message send time and the local PE PEDynamic table. So the recipient PE can update its PEDynamic table and calculate the latency with the sending PE.

The work location mechanism in *GRID-GUM2* has the advantage of encouraging PEs to obtain work and hence data from PEs that currently have low-latency communication. It also has the advantage of changing the behaviour of PEs to be less aggressive for seeking and accumulating work from PEs that have faster CPU speed. That means, PEs with slow CPU speed source work from PEs have the same slow CPU speed or from PEs with fast CPU speed but they are heavily loaded relative to their CPU speed. As a result this work location mechanism

decreases the total amount of communication and specially through high-latency communication which certainly improves the performance.

4.5.3 Work Request Handling Mechanism

The work request handling mechanism in *GRID-GUM*1 follows a naive policy. As shown in Figure 4.20, if a PE receives a FISH message, it searches for a spark in the spark pool and, if available, sends the spark to the PE in which the FISH message was originally generated in a SCHEDULE message. If the spark pool is empty and the FISH message has not exceeded its age limit, the recipient PE forwards the FISH message randomly to another PE. Otherwise the FISH message is returned back to the original PE.

```
IF received FISH THEN
  IF sparks available THEN
    send spark in SCHEDULE to originPE
  ELSE
    IF FISH exceeded age THEN
      return back to originPE
    ELSE
      send FISH to random PE
    FI
  FI
FI
```

Figure 4.20: *GRID-GUM*1 Work Request Handling Algorithm

As discussed in Section 4.2, *GRID-GUM*1's blind and strictly passive mechanism of work request handling is based on either returning constantly a single spark or forwarding the FISH message randomly to another PE. In addition to the disadvantages thar are from forwarding the fish message randomly to another PE, one can distinguish two more non-scalable scenarios caused by returning constantly a single spark:

- From Figure 4.13 and Table 4.8, `PE9` in `cluster C`, which is idle, sends a FISH message to `PE11` in `cluster D` which is heavily loaded. `PE9` and `PE11`

reside in two different clusters and are connected via high-latency communication. In addition to that, `PE9` has faster CPU speed than `PE11` and the total CPU speed of `cluster C` is greater than total CPU speed of `cluster D`. Furthermore, PEs in `cluster C` likely to be lightly loaded; `cluster C` is idle. If `PE11` sends a SCHEDULE message with a single spark, which is the case in *GRID-GUM*1, to `PE9`, `cluster C` will remain idle and PEs reside in `cluster C` has to obtain work remotely from outside the cluster via high-latency communication. If all PEs in `cluster C` obtain work remotely through high-latency communication that will certainly impinge upon the total performance.

- Conversely to the previous case, if `PE11` in `cluster D`, which is idle in this case, sends a FISH message to `PE9` in `cluster C`. `PE9` has spare spark(s) in the spark pool, but it is not heavily loaded to its CPU speed. Under *GRID-GUM*1 work request handling mechanism, `PE9` sends a SCHEDULE message with a spark to `PE11`. This scheduling decision degrades performance for two reasons:

  - the spark may be fished back by another PE in `cluster C` before `PE11` starts activating it due its slow performance, and that requires extra communication via high-latency communication which have a non-scalable impact in the performance, or
  - `PE11` might activate the spark directly after it received it, and all this happen at the end of the program, this means all PEs might have to wait until `PE11` had finished the evaluation, which might take longer than if the spark has not been fished from `PE9`.

The work request handling in *GRID-GUM*2 introduces a new mechanism to minimise the intra-cluster (high-latency) communications. Figure 4.21 introduces the work request handling mechanism in *GRID-GUM*2 in pseudocode level, the bold font is used to show the new adaptive load distribution.

Work request handling mechanism in *GRID-GUM*2 operates as follow:

At the beginning, the recipient PE profits from the message send time and PEDynamic table attached to the message to calculate latency with the sending PE and updates its PEDynamic table with recent load information. The recipient PE uses the load and the CPU speed information from PEDynamic and PEStatic tables respectively to calculate the relative load to the CPU speed for both the local PE (localRatio) and the original PE (originRatio), and this comparison is due to the possibility the FISH message has been forward from another PE different than the original one. If the localRatio appears to be less than the originRatio, that means the recipient PE is heavily loaded relative to its CPU speed. In this case, if the recipient PE and the original PE reside in two different clusters, according to the clusterId in the PEStatic table, the recipient PE examines the relative load for local cluster (localClusterRatio) and the original cluster (originClusterRatio). If the localClusterRatio appears to be greater than originClusterRatio, the recipient PE considers itself and the original PE as they reside in the same cluster and sends a SCHEDULE message with a single spark to the original PE. Conversely, if the originClusterRatio is greater than localClusterRatio, the recipient PE splits the local work with the original PE according to the total CPU speeds in both original cluster and local cluster. Under these circumstances, the number of sparks are sent with the SCHEDULE message is the ratio of the half local load multiply by the total CPU speeds of the original cluster, to the summation of total CPU speeds in the original cluster and the local cluster, unless the result of this ratio is less than one, the recipient PE sends a SCHEDULE message with a single spark. In the other cases where the local PE does not have work to donate, or the localRatio is greater than the originRatio, the recipient PE examines the FISH message age, if it has not exceeded its age limit, then it forwards the FISH message, with send message time and local PEDynamic table, using the work allocation mechanism in *GRID-GUM2*, to another PE. However, if the FISH message has exceeded its age limit, the local PE returns the unsuccessful FISH to the original PE.

To summarise, the work request handling mechanism in *GRID-GUM2* is based on using the static and dynamic information about the recipient and the original

PEs and their clusters to deal with cases where is no work available in the cluster of the original PE. The important point resides in switching from passive to active mode by involving the recipient PE in the scheduling decision, this shows that *GRID-GUM2* does not follow a strict passive load distribution. The recipient PE has to examine the capabilities and the loads of the original cluster and compare it with the local cluster capabilities and loads. Broadly speaking, the work request handling mechanise in *GRID-GUM2* distinguish between two scenarios:

- if the FISH message originated from relatively powerful cluster, then multiple sparks are returned in the SCHEDULE message;
- if the FISH message originated from a relatively weak cluster then the request is served as usual, i.e. as a single spark.

4.6 Summary

GRID-GUM2's adaptive load distribution mechanism may be summarised thus:

- Idle PEs aim to obtain work from a PE that is heavily loaded.
- A PE only sends work to a fishing PE only if its relative load less than the fishing PE.
- Idle PEs prefer to obtain work from PEs that currently have low-latency communication.
- A PE switches from a passive mode to an active mode when it receives a FISH message out of the cluster.
- The main PE is nominated at the beginning of the execution as the fastest PE within the biggest cluster.

```
IF received FISH THEN
  update ComMap and PEDynamic Tables with FISH's latency
   and load data
  IF sparks available THEN
   calculate localRatio, originRatio
   IF originRatio > localRatio THEN
     IF originPE and localPE has same clusterId THEN
       send spark in SCHEDULE to originPE + Dynamic Info.
     ELSE
       calculate localClusterRatio and originClusterRatio
       IF originClusterRatio > localClusterRatio THEN
         calculate noOfSparksToSend
       FI
       IF noOfSparksToSend < 1 OR
        originClusterRatio < localClusterRatio THEN
         send spark in SCHEDULE to originPE + Dynamic Info.
       ELSE
         send (noOfSparksToSend) spark(s) to originPE
           +Dynamic Info.
       FI
     FI
   FI
  ELSIF originRatio < localRatio THEN
   IF FISH exceeded age THEN
     return back to originPE + Dynamic Info.
   ELSE
     FOR each PE in comMap Table
       calculate destRatio
       IF originRatio > destRatio THEN
        send FISH to destPE + Dynamic Info.
         break
       FI
     end FOR
     IF NOT mainPE AND NO FISH message sent THEN
      send FISH to mainPE + Dynamic Info.
    FI
   FI
  FI
FI
```

Figure 4.21: *GRID-GUM2* work request handling algorithm

Chapter 5

GRID-GUM2 Evaluation

5.1 Introduction

This Chapter evaluates the performance of the new adaptive load distribution mechanism, *GRID-GUM2*, on a range of computational Grid configurations:

On relatively low-latency communications, we measure performance on:

- homogeneous architectures
- heterogeneous architecture

On high-latency communications, we measure performance on:

- homogeneous architecture
- heterogeneous architecture

Finally, we measure *GRID-GUM2*'s scalability on a high-latency heterogeneous architecture.

5.1.1 *GRID-GUM1.1*

A special implementation of *GRID-GUM2*, *GRID-GUM1.1*, is used in this evaluation to study the contribution of the static information, CPU speed, on performance. *GRID-GUM1.1* uses CPU speed information to choose the mainPE as

the PE with the fastest CPU speed. Moreover, it uses the CPU speed information to prevent PEs with slow CPU speed from extracting work from PEs with faster CPU speed unless the latter is the mainPE. Unlike *GRID-GUM2*, *GRID-GUM1.1* does not collect or employ the loads and latencies information. CPU speed information is passed to *GRID-GUM1.1* through an external file which is provided by the user.

5.2 Measurement Framework

5.2.1 Hardware Apparatus

The measurements have been performed on five Beowulfs clusters: three located at Heriot-Watt Riccarton campus (*Edin1, Edin2,* and *Edin3*), a cluster located at Ludwig-Maximilians University Munich (*Muni*), and a cluster located at Heriot-Watt Borders campus(*SBC*); see Tables 5.9 and 5.10 for the characteristic of these Beowulfs.

Beowulfs	CPU	Cache	Memory	PEs
	Speed MHz	kB	Total kB	
Edin1	534	128	254856	32
Edin2	1395	256	191164	6
Edin3	1816	512	247816	10
Muni	1529	256	515500	7
SBC	933	256	110292	4

Table 5.9: Characteristics of Beowulf Clusters

	Edin1	Edin2	Edin3	SBC	Muni
Edin1	0.20	0.27	0.35	2.03	35.8
Edin2	0.27	0.15	0.20	2.03	35.8
Edin3	0.35	0.20	0.20	2.03	35.8
SBC	2.03	2.03	2.03	0.15	32.8
Muni	35.8	35.8	35.8	32.8	0.13

Table 5.10: Approximate Latency Between Clusters (ms)

5.2.2 Software Apparatus

Six programs are measured in this experiment. The `parFib` computes Fibonacci numbers. The `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. The `queens` program places chess pieces on a board. The `raytracer` calculates a 2D image of a given scene of 3D objects by tracing all rays in a given scene of 3D objects by tracing all rays in a given grid, or window. The `matMult` multiples two matrices. The `linSolv` program finds an exact solution of a linear system of equations. For more details about these programs see Appendix A.2

Program	Application Area	Paradigm	Regularity
`queens`	Hustric Search	Div-Conq.	Regular
`parFib`	Numeric	Div-Conq.	Regular
`linSolv`	Symbolic algebra	Data Par.	Limit irreg.
`sumEuler`	Numeric Analysis	Data Par.	Irregular
`matMult`	Numeric	Div-Conq.	Irregular
`raytracer`	Graphic	Data Par.	High irreg.

Table 5.11: Programs Characteristics and Performance

Three programs have regular parallelism `queens`, `parFib` and `matMult`; three programs have irregular parallelism `sumEuler`, `linSolv` and `raytracer`. Programs with regular parallelism generate threads which have approximately the same cost of computation. Programs with irregular parallelism generate threads which require different cost of computation. Moreover, irregular-parallel programs generate threads at different stages through the course of execution. Among these six programs, `queens`, `sumEuler` and `linSolv` have relatively low-communication degree, and `parFib`, `matMult` and `raytracer` have relatively high-communication degree, Table 5.11.

5.3 Low-Latency Computational GRID

5.3.1 Low-Latency: Heterogeneous Performance

First we investigate the performance impact of using the adaptive load distribution of *GRID-GUM2* on heterogeneous computational architecture with moderate latency communication.

The measurements in Table 5.12 use *GRID-GUM1* and *GRID-GUM2* on 4 PEs from Edin1 and and 4 PEs from Edin2 Beowulf In the table, the second and third columns record the run-time in seconds using *GRID-GUM1* and *GRID-GUM2* respectively. The last column shows the percentage improvement of *GRID-GUM2*. All run-times in this experiment represent the median of three executions to ameliorate the impact of operating system and shared network interaction.

Program	Run-time (s)		Improvement %
	GRID-GUM1	*GRID-GUM2*	
`raytracer`	1340	572	57%
`queens`	668	310	53%
`sumEuler`	570	279	51%
`linSolv`	217	180	17%
`matMult`	94	86	9%
`parFib`	136	134	1%

Table 5.12: Performance on Heterogeneous Architecture

Measurements in Table 5.12 show that *GRID-GUM2* outperforms *GRID-GUM1* on multiple heterogeneous clusters with moderate latency communication as far as the execution time is concerned. However, these measurements indicate a poor improvement with `parFib` and `matMult`. This is due to the characteristics of these programs. `parFib` is a perfect program. It has high levels of parallelism and a low-communication degree. Thus, `parFib` manages heterogeneity under *GRID-GUM1* without the need of an adaptive load distribution, which leaves *GRID-GUM2* with no possibility of improvement. In contrast, `matMult` is not a perfect parallel program, scaling only up to 4 PEs [LRS+03]. This is due to the

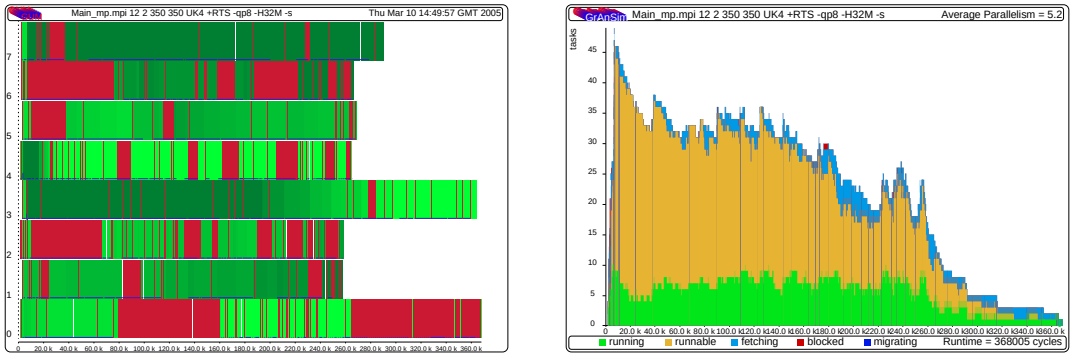
limited parallelism generated from the program. Thus, `matMult` under *GRID-GUM2* has a modest improvement of 9%. *GRID-GUM2* selects the 4 fast PEs from Edin2 Beowulf cluster to carry the computation. However, due to the lack of parallelism, the 4 Edin1 PEs remain with no work, thus, the possible improvement of `matMult` under *GRID-GUM2* is limited.

`linSolv` scores a modest improvement under *GRID-GUM2* of 17%. The limited irregular parallelism and the low-communication degree in `linSolv` helps *GRID-GUM1* overcome the heterogeneous architecture without an adaptive load distribution mechanism. Due to this, the gains from using the adaptive load distribution of *GRID-GUM2* to improve `linSolv` is partially limited.

The programs `raytracer`, `queens` and `sumEuler` show the highest improvement under *GRID-GUM2*; more than 50%. This is because of the low degree of parallelism, which means the small number of threads, in these programs which make them more sensitive to a heterogeneous architecture and dependable on an adaptive load distribution to score a good improvement. Through the rest of this sub-section we study in more details the behaviour of `raytracer` heterogeneous computational GRID architecture, due to its high irregular parallelism and high-communication degree.

`raytracer`, as explained before, has highly irregular execution, and consequently is very sensitive to changes in parallel environment. Figure 5.221 shows per-PE and overall activity profiles for `raytracer`, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). A per-PE activity profile shows the behaviour for each of the PEs (y-axis) over execution time (x-axis), as described in Section B.2. An overall activity profile shows the behaviour of the program at each instant of its execution, as described in Section B.1.

Figure 5.22.a shows a poor load distribution of *GRID-GUM1* with `raytracer` to calculate an image with resolution $350X350$. PEs have numerous idle period and finish at different time. From Figure 5.22.b, it is observable that there are a considerable number of runnable threads waiting to be evaluated for most of the execution time. This may explain the poor load distribution in *GRID-GUM1*.



a) Per-PE activity profile

b) Overall activity profile

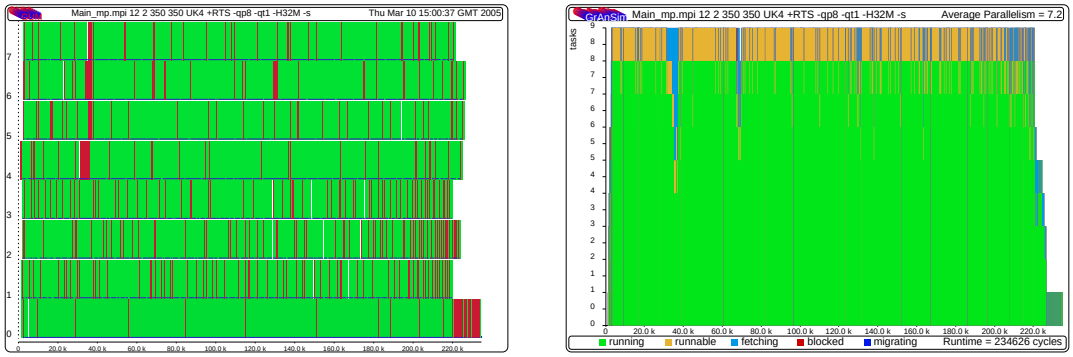
Figure 5.22: *GRID-GUM*1: `raytracer` with $350X350$ Image on a Heterogeneous Computational Grid

PEs with slow CPU speed in a heterogeneous architecture in *GRID-GUM*1 show the same demand of seeking work as PEs with fast CPU speed. This suggests that PEs with slow CPU speed accumulate and activate sparks at the PEs have fast CPU speed. If a spark has been activated, it remains in its local PE as a runnable or blocked thread in the thread pool and it can not be evaluated by another PE. PEs have different capabilities of evaluating their own threads so many runnable threads are waiting to be evaluated while some PEs are idle.

*GRID-GUM*1 provides explicit control over the load distribution by specifying a hard limit on the total number of live threads, i.e. runnable or blocked threads. Figure 5.23 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution $350X350$. *GRID-GUM*1 in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.

In Figure 5.23, *GRID-GUM*1 with thread limitation shows an efficient load distribution in a heterogeneous architecture with a moderate latency communication. It completes the image manipulation in 327 s, while the version of *GRID-GUM*1 which does not employ thread limitation requires 441 s. As expected for the same problem *GRID-GUM*2 has similar performance, i.e. 338 s, with *GRID-GUM*1 using thread limitation as depicted in Figure 5.24.

However, *GRID-GUM*1's load distribution efficiency declines when the size



a) Per-PE activity profile *b*) Overall activity profile

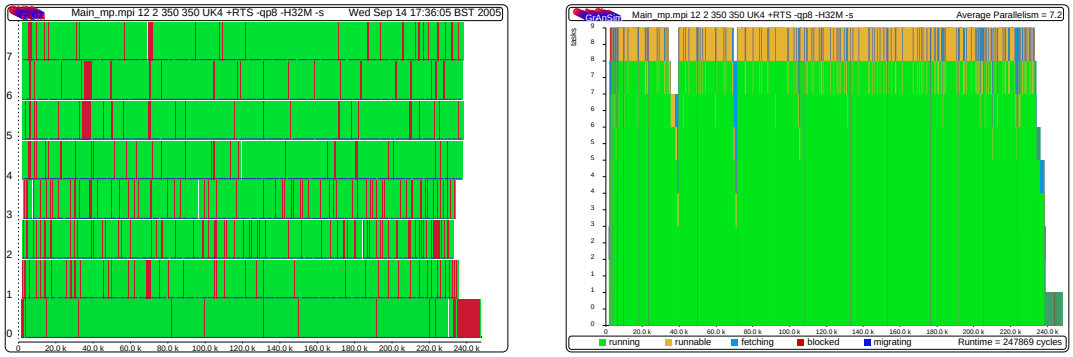
Figure 5.23: *GRID-GUM*1 with Thread Limitation: `raytracer` with $350X350$ Image on Heterogeneous Computational Grid

of the input increases even with thread limitation. Figure 5.25 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution $500X500$. *GRID-GUM*1 in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.

The PEs in Figure 5.25.a finish at the same time, but they still have numerous idle periods which reduces performance. These idle periods are caused by the dependencies between threads in `raytracer`. These dependencies are affected badly by the thread limitation, which causes PEs to remain idle waiting for certain threads to be evaluated. Figure 5.25.b shows that the idle periods are not caused by lack of tasks to be evaluated. Generally speaking, thread limitation has a serious impact on many programs' performance.

Figure 5.26 shows per-PE profiles for `linSolv` with and without thread limitation on 8 homogeneous machines from Edin1 Beowulf cluster. *GRID-GUM*1 delivers better performance with `linSolv` without using thread limitation. *GRID-GUM*1 requires 2802 s to finish `linSolv` computation using thread limitation, but when thread limitation is excluded *GRID-GUM*1 requires only 1521 s is required.

However, *GRID-GUM*2 shows more effective load distribution in a heterogeneous architecture in comparison with *GRID-GUM*1. Figure 5.27 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution



a) Per-PE activity profile

b) Overall activity profile

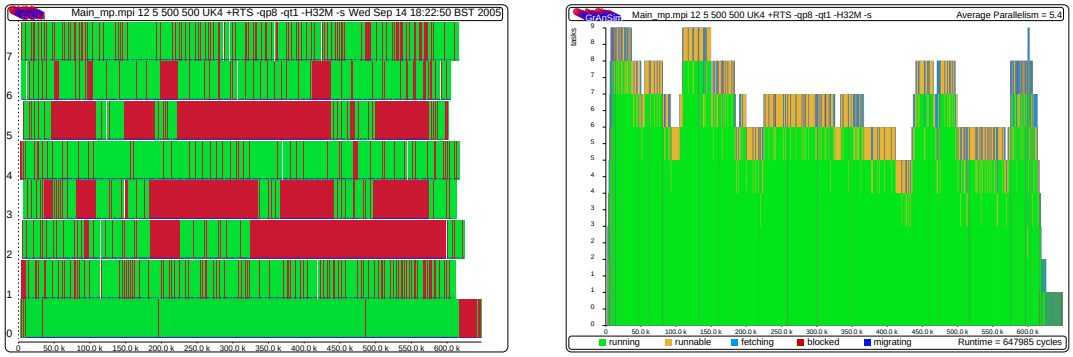
Figure 5.24: *GRID-GUM2*: `raytracer` with $350X350$ Image on Heterogeneous Computational Grid

$500X500$.

PEs in Figure 5.27.a are fairly balanced and finish at about the same time. Figure 5.27.b shows that *GRID-GUM2* scores good average parallelism in 8 PEs, 6.9, and generates enough tasks for all PEs at each instant of the execution time. Finally, *GRID-GUM2* outperforms *GRID-GUM1* with thread limitation in `raytracer` when the image resolution increases from $350X350$ to $500X500$. The execution time for *GRID-GUM2* and *GRID-GUM1* with thread limitation is 572 s and 814 s, respectively.

Summary

- *GRID-GUM2* effectively manages the parallel execution of our test programs on heterogeneous computational Grid architecture (Table 5.12);
- Thread limitation improves the performance of *GRID-GUM1* on heterogeneous computational Grid architecture for most of our test programs (Figures 5.23 and 5.26);
- *GRID-GUM2* outperforms *GRID-GUM1* regardless of the thread limitation especially when the input size is relatively big (Figures 5.25 and 5.27).



a) Per-PE activity profile

b) Overall activity profile

Figure 5.25: *GRID-GUM*1 with Thread Limitation: `raytracer` with $500X500$ Image on Heterogeneous Computational Grid

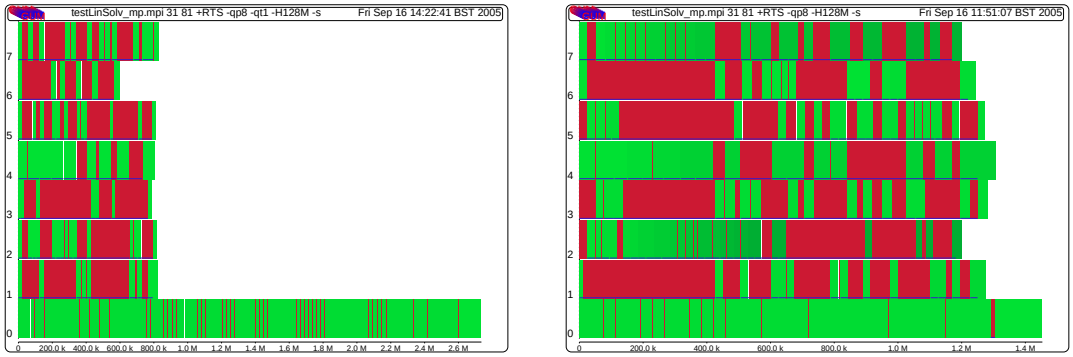
5.3.2 Low-Latency: Homogeneous Performance

This experiment investigates the performance impact of the adaptive load distribution of *GRID-GUM*2 on a low-latency homogeneous computational Grid architecture.

Variability

The measurements in Table 5.13 have been performed on 10 PEs from the Edin1 cluster. In Table 5.13, the second and fifth columns record the mean of 50 runs in seconds. The third and sixth columns show the variance of the 50 runs. The fourth and seventh columns present the percentage variance relative to the mean. The last column shows the percentage improvement in *GRID-GUM*2's percentage ratio between variance and mean to *GRID-GUM*1's percentage ratio between variance and mean.

Table 5.13 shows that *GRID-GUM*1 has unpredictable variance in the results especially with programs with irregular parallelism. In Table 5.13, the programs with regular parallelism show less percentage variation, 23% in `queens` and 26.7% in `parFib`. However, the programs with irregular parallelism show a higher percentage ratio between the variance and mean, 47% in `sumEuler`,



a) With thread limitation *b*) With no thread limitation

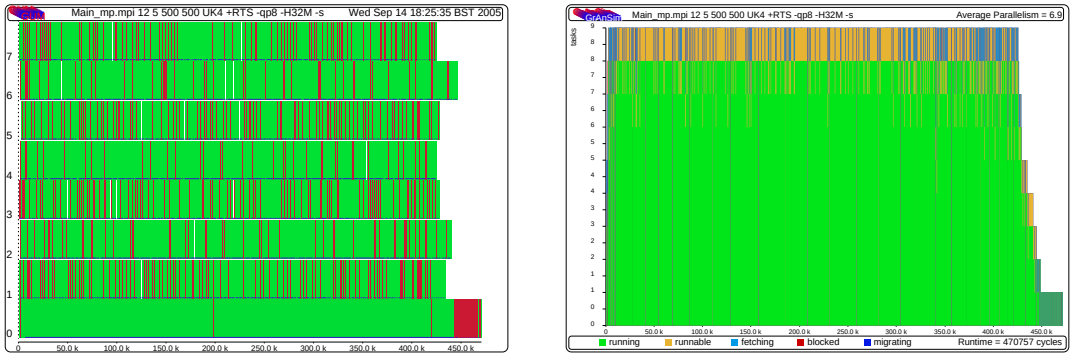
Figure 5.26: *GRID-GUM*1: `linSolv` on Heterogeneous Computational Grid

Program Name	*GRID-GUM*1			*GRID-GUM*2			Impr%
	Mean Rtime(s)	Var	Var%	Mean Rtime(s)	Var	Var%	
`queens`	648.97	149.9	23.0%	649.59	2.62	0.4%	98%
`parFib`	84.91	22.68	26.7%	88.85	3.65	4.1%	84%
`linSolv`	176.21	63.82	36.2%	149.82	7.20	4.8%	86%
`sumEuler`	117.82	55.43	47.0%	116.28	20.33	17.4%	63%
`raytracer`	476.93	168.15	35.2%	448.53	27.93	6.2%	82%

Table 5.13: Variation Between *GRID-GUM*1 and *GRID-GUM*2 on 10 PEs

36.2% in `linSolv` and 35.2% in `raytracer`. The unpredictable behaviour in *GRID-GUM*1 is due to its load distribution mechanism which is based on a naive blind fishing mechanism, as explained in Section 4.2. If the idle PE in *GRID-GUM*1 chooses randomly the right PE to donate work, the performance certainly improves; in contrast, if the idle PE picks the wrong PE to donate work, the performance deteriorates. So it all depends on the random choice of the fishing PE which caused the wide variance in the program behaviour.

However, *GRID-GUM*2 shows less variable behaviour in comparison with *GRID-GUM*1's behaviour. In Table 5.13, `queens` and `parFib` show an improvement of 98% and 84% respectively. `sumEuler`, `linSolv` and `raytracer`, which have irregular parallelism, show improvements of 63%, 66% and 82% respectively.



a) Per-PE activity profile

b) Overall activity profile

Figure 5.27: *GRID-GUM2*: `raytracer` with $500X500$ Image on Heterogeneous Computational Grid

It is notably that `raytracer` and `linSolv` have less variable performance than `sumEuler`. That is due to, `raytracer` and `linSolv` running for a longer time than `sumEuler` which more fully up data the load information in *GRID-GUM2*. Moreover, the parallel implementation of `raytracer` is based on generating most of the parallelism at the beginning of the execution.

Overheads

The second part of the the measurements in this section are presented in Table 5.14. These measurements have been performed on 16 PEs from Edin1 Beowulf cluster. Moreover, all runtimes in this experiment represent the median of three executions, to ameliorate the impact of operating system and shared network interaction. The first and second columns show the program name and the different runtime environment implementation. In the second column **GG**1 and **GG**2 stand for *GRID-GUM*1 and *GRID-GUM*2 respectively. The third column records the total number of threads generated during the execution. The remaining columns show averages over all processors for the maximal heap residency (i.e. the maximum amount of heap that is alive at garbage collection time) the allocation rate (i.e. the amount of local memory allocated per second of execution time) the communication degree (i.e. the number of massages sent per second of

execution time) and the average packet size (i.e. the size of packet in Byte).

Program Name	RTE	No of Threads	Max Heap Residency (KB)	Alloc Rate (MB/s)	Comm Degree (Msgs/s)	Average Pkt Size (Byte)
parFib	GG1	26595	5.12	55.3	15.55	5.58
	GG2	26595	5.12	43.2	14.87	5.60
sumEuler ‘	GG1	82	62.4	52.8	2.09	90.26
	GG2	82	62.4	45.7	0.73	90.28
raytracer	GG1	350	538.6	60.0	62.72	321.75
	GG2	350	538.6	49.5	46.93	323.00
linSolv	GG1	242	437.2	40.3	5.50	290.67
	GG2	242	437.2	26.5	2.54	276.37
matMult	GG1	144	4.3	39.0	67.30	208.85
	GG2	144	4.3	40.0	31.29	209.37
queens	GG1	24	2.03	38.8	0.26	851.85
	GG2	24	2.03	34.0	0.13	846.17

Table 5.14: Dynamic Program Properties on 16 PEs

Table 5.14 shows that the adaptive load distribution of *GRID-GUM2* decreases the communication degree. This is due to the dynamic and static information about load, latencies and CPU speed which are provided by *GRID-GUM2*. This information assists the programs to accomplish the computation without exchanging many messages. However, the rest of the programs' dynamic properties remain unchanged. For instance, the number of threads and the maximum heap residency remain without any change between *GRID-GUM1* and *GRID-GUM2*. The allocation rates and average packet sizes have very minor changes. These are due to the impact of the operating system and the shared network. In short, the new adaptive load distribution of *GRID-GUM2* acts in an independent way of collecting and maintaining the dynamic and static information. Due to this the programs have the same thread and packet behaviour without any changes in the dynamic properties.

Summary

- Programs under *GRID-GUM2* have less variance in the performance than under *GRID-GUM1* (Table 5.13).
- *GRID-GUM2* retains a very light overhead which does not effect the program's dynamic properties (Table 5.14).

The measurements in both this section and in Chapter 3 clearly show that, among all our benchmark programs only `raytracer`, `sumEuler` and `queens` exhibit sensitivity to high-latency and heterogeneous computational Grid architectures. Due to this, the experiment in the following sections (Sections 5.4 and 5.5) throughly investigate the behaviour of those programs under different computational Grid architectures.

5.4 High-Latency Computational Grid

This section study the performance impact of the adaptive load distribution of *GRID-GUM2* on homogeneous and heterogeneous computational Grid architectures with high-latency network.

All run-times in this experiment represent the median of three executions to ameliorate the impact of operating system and shared network interaction.

5.4.1 High-Latency: Heterogeneous Performance

The measurements in Tables 5.15 (`raytracer`), 5.16 (`sumEuler`) and 5.17 (`queens`) are performed on two heterogeneous Beowulf clusters, Edin1 and Muni. PEs in the Edin1 Beowulf cluster have slower CPU speed than those in the Muni Beowulf cluster. Moreover, Edin1 and Muni Beowulf clusters are connected over a high-latency communication, for more details see Tables 5.9 and 5.10.

In Tables 5.15, 5.16 and 5.17, each Edin1 machine is labelled E and each Muni machine is labelled M. The first and second columns show case number and different combination of PEs from Edin1 and Muni Beowulf clusters respectively.

The third, fourth and sixth columns record the run-time in seconds for *GRID-GUM*1 (*GG*1), *GRID-GUM*1.1 (*GG*1.1) and *GRID-GUM*2 (*GG*2) respectively. The fifth column shows the static information (CPU speed) contribution to the performance change under *GRID-GUM*1.1 in comparison with *GRID-GUM*1. The seventh column indicates the dynamic information (loads and latencies) contribution to the change under *GRID-GUM*2. The last column reports the total performance change of using static and dynamic information in the adaptive load distribution of *GRID-GUM*2 in comparison with *GRID-GUM*1.

raytracer

Case		GG1	GG1.1	Static Impr	GG2	Dynamic Impr	Total Impr
1	1E3M	1658	748	54%	716	2%	56%
2	1E4M	1490	689	53%	583	7%	60%
3	1E2M	1607	975	39%	848	8%	47%
4	2E3M	1223	745	39%	716	2%	41%
5	2E2M	1396	965	30%	909	4%	34%
6	3E2M	1254	983	21%	961	2%	23%
7	1E1M	1934	1678	13%	1689	0%	13%
8	2E1M	1778	1687	5%	1326	20%	25%
9	3E1M	1495	1832	-22%	1305	34%	12%
10	4E1M	1296	1597	-23%	1236	27%	4%

Table 5.15: `raytracer`: Heterogeneous High-Latency Computational Grid

In Table 5.15, *GRID-GUM*1.1 (*GG*1.1) shows a substantial improvement in the cases (1, 2, 3, 4) where the number of fast PEs (M) is more than the number of slower PEs (E).

For instance, in case 1 *GRID-GUM*1.1 decreases the run-time required to finish the `raytracer` computation in four PEs ($1E3M$), one local E PE and three remote M PEs, from 1658 s to 748 s, with improvement of 54%. However, the behaviour become worst under *GRID-GUM*1.1 if the number of PEs with slow CPU speed (E) is greater than the number of PEs with faster CPU speed (M),

cases (6, 8, 9, 10).

For instance, in case 10 GRID-GUM1.1 increases the run-time required to finish the `raytracer` computation in five PEs ($4E1M$), four local E PEs and one remote M PE, from 1296 s to 1597 s, with a degradation of -23%. This behaviour of `raytracer` under GRID-GUM1.1 is related to the high-latency communication. In a configuration of the form ($xEyM$), where $x > y$, cases (6, 8, 9, 10), GRID-GUM1.1 nominates the mainPE from M PEs. In this case, E PEs have to seek work during the course of the execution from M PE(s), with no restriction, through high-latency communication. In programs like `raytracer` with relatively high-communication degree, a high-latency communication has a major impact on GRID-GUM1.1 performance due to the number of messages that are exchanged over the high-latency communication, as shown in the cases 10 ($4E1M$), 9 ($3E1M$) and 8 ($2E1M$).

As depicted in the seventh column in Table 5.15, use of dynamic information of loads and latencies produced by GRID-GUM2 inspires the performance improvement. However, this improvement varies according to the number of remote and local PEs and their CPU speed. In a configuration of the form ($xEyM$), where $x < y$, dynamic information has a limited contribution to the performance.

For instance, in case 2 to finish the computation of `raytracer` in five PEs ($1E4M$), one local E PE and four remote M PEs, GRID-GUM1 requires 1490 s. Using only static information in the adaptive load distribution helps to improve the performance with the same configuration as GRID-GUM1.1 by 53%. The dynamic information contributes little improvement, 7%. Under this configuration ($1E4M$), GRID-GUM2 nominates the mainPE from the M PEs. That means there is only one PE which needs to seek work remotely over a high-latency communication. Moreover, this PE has a relatively slow CPU speed which means it requires less work to keep busy. Thus, this PE exchanges fewer messages especially over a high-latency communication. In contrast, for configurations of the form ($xEyM$), where $x > y$, the dynamic information has a greater contribution to the performance. For instance, in case 9 to finish the computation of `raytracer` with three local PEs with slow CPU speed and one remote PE

with faster CPU speed ($3E1M$), *GRID-GUM*1 requires 1495 s. Using only the static information under *GRID-GUM*1.1, the performance deteriorates by 22%. However, the dynamic information contributes by improving the performance overall by 34%. The dynamic information in this case helps the adaptive load distribution of *GRID-GUM*2 to nominate the mainPE from among the E PEs, decreasing the number of PEs which are required to seek work over a high-latency communication. In addition to that, *GRID-GUM*2 uses the loads information to transfer enough work to a M PE, to decrease the number of messages over a high-latency communication. In this case, the dynamic information in the adaptive load distribution demonstrates an important role in improving the performance.

Broadly speaking, the static and dynamic information of the adaptive load distribution of *GRID-GUM*2 seems essential to bring an improvement of the performance of a program with relatively high-communication degree and irregular parallelism like `raytracer`. For instance, in case 2, to finish the computation of `raytracer` in five PEs ($1E4M$), *GRID-GUM*1 requires 1490 s. However, *GRID-GUM*2 requires only 583 s, an improvement of 60%.

sumEuler

Case		GG1	GG1.1	Static Impr	GG2	Dynamic Impr	Total Impr
1	2E2M	247	181	27%	168	5%	32%
2	1E4M	183	144	21%	131	7%	28%
3	1E1M	480	384	20%	351	7%	27%
4	1E3M	189	151	20%	149	1%	21%
5	2E3M	229	199	13%	173	11%	24%
6	1E2M	214	195	9%	175	9%	18%
7	3E2M	219	223	-2%	189	16%	14%
8	3E1M	306	318	-4%	282	11%	8%
9	2E1M	305	326	-6%	292	10%	4%
10	4E1M	305	325	-6%	290	10%	5%

Table 5.16: `sumEuler`: Heterogeneous High-Latency Computational Grid

In Table 5.16, the performance of `sumEuler` under *GRID-GUM*1.1 (*GG*1.1) shows a major improvement in the cases where the number of fast M PEs is more or equal than the number of slower E PEs.

For instance, in case 1 *GRID-GUM*1.1 decreases the run-time required to finish the `sumEuler` computation in four PEs ($2E2M$), two local PEs with slow CPU speed and two PEs with faster CPU speed, from 247 s to 181 s, with improvement of 27%. *GRID-GUM*1.1 nominates the mainPE from M PEs. In this case, *GRID-GUM*1.1 benefits from M PEs with fast CPU speed to improve the performance of `sumEuler`, due to all work being located in the M PEs side. Thus, M PEs can quickly accumulate enough work before remote E PEs with slow CPU speed start fishing. In addition, E PEs seek work remotely over high-latency communication. However, the impact of this is negligible due to `sumEuler`'s low-communication degree. However, *GRID-GUM*1.1 progress declines when the number of E PEs with slow CPU speed increases in comparison with the number of M PEs with fast CPU speed.

For instance, in case 10 *GRID-GUM*1.1 increases the run-time required to finish the `sumEuler` computation in five PEs ($4E1M$), four local PEs with slow CPU speed and one remote PE with faster CPU speed, from 305 s to 325 s, with a negative improvement of -6%. *GRID-GUM*1.1 results have been detrimentally effected by the irregular parallelism of `sumEuler` especially if the number of PEs with slow CPU speed is greater than the number of PEs with fast CPU speed.

In a configuration of the form, ($xEyM$), where $x > y$, *GRID-GUM*1.1 blindly distributes the work equally between E PEs and M PE(s). Due to the irregular parallelism in `sumEuler`, the tasks distributed between E PEs and M PEs have different costs of computation. Moreover, those tasks generate other tasks at different stages through the course of execution. *GRID-GUM*1.1's uninformed load distribution can not assist M PEs with fast CPU speed to send the fish messages to the right PE. In addition, E PEs act as aggressively as M PEs at the beginning to accumulate work. This leaves E PEs heavily loaded in relation to their CPU speed. The main drawback to this situation under *GRID-GUM*1.1 is that M PEs can not find the correct destination for its fish messages. Moreover,

the E PEs might activate all the tasks are received to threads which prevent other PEs from having access to those tasks.

In the seventh column in Table 5.16, the dynamic information of loads and latencies shows a positive impact on the performance. The improvement of using the dynamic information with `sumEuler` lies in a small range between 1% and 16%, and this is due to `sumEuler`'s program properties. `sumEuler` has a relatively low-communication degree, which demonstrates the limitations of latencies information. However, irregular parallelism of `sumEuler` supports the need of using the dynamic information of loads to improve the performance especially when there are groups of PEs of slow CPU speed and fast CPU speed. As depicted in the Table 5.16 the dynamic information improves the performances by 16% in comparison to *GRID-GUM*1.1 in case 7 using five PEs ($3E2M$), three local E PEs and two remote M PEs.

In general, the adaptive load distribution of *GRID-GUM*2, shows a major improvement on `sumEuler` performance, 32%. *GRID-GUM*2 employs both CPU speed, static information, and loads and latencies, dynamic information, to distribute works between PEs according to the load and CPU speed for each PE and the number of PEs can communicate with them through a low-latency communication. For instance, `sumEuler` under *GRID-GUM*2 requires only 168 s to finish the computation which requires under *GRID-GUM*1 247 s using four PEs ($2E2M$), two local PEs with slow CPU speed and two remote PEs with fast CPU speed. In this case, *GRID-GUM*2 allows the $2E$ PEs to accumulate enough work regarding their CPU speed and the total available amount of work. Thus, the $2M$ PEs have enough work to keep them busy until the end of the computation.

However, in some cases *GRID-GUM*2 shows only a modest improvement against *GRID-GUM*1. For instance, in case 10 `sumEuler` under *GRID-GUM*2 requires 290 s to finish the computation which requires 305 s under *GRID-GUM*1 using five PEs ($4E1M$), four local PEs and one remote PE. This leaves *GRID-GUM*2 with 4% improvement. In this case, *GRID-GUM*2 favours the four local E PEs to finish the computation. Thus, it nominates the mainPE from the E PEs and grands the remote M PE with enough work to keep it busy until the

end of the computation. This strategy helps to decrease the communication over high-latency communication, but has the drawback of isolating the M PE. M PE could finish its computation earlier, however, due to the lazy passive mechanism of exchanging the dynamic information, the M PE assumes that other E PEs can not contribute to its request of work according to the available loads information. Due to this, the M PEs contribution to improve the performance is decreased.

queens

Case		GG1	GG1.1	Static Impr	GG2	Dynamic Impr	Total Impr
1	1E4M	421	324	23%	295	6%	29%
2	2E3M	536	412	23%	346	12%	35%
3	1E3M	573	444	22%	388	10%	32%
4	1E2M	614	482	21%	451	5%	26%
5	3E2M	621	495	20%	465	5%	25%
6	2E2M	630	500	20%	460	6%	26%
7	3E1M	684	560	18%	498	9%	27%
8	4E1M	632	530	16%	454	12%	28%
9	2E1M	789	659	16%	639	3%	19%
10	1E1M	821	699	14%	665	5%	19%

Table 5.17: `queens`: Heterogeneous High-Latency Computational Grid

In Table 5.17, *GRID-GUM*1.1 ($GG1.1$) shows an improvement in all cases.

For instance in case 1 *GRID-GUM*1.1 shows some improvement against *GRID-GUM*1. It decreases the run-time required to finish the `queens` computation in five PEs ($1E4M$), one local PE with slow CPU speed and four remote PEs with faster CPU speed, from 421 s to 324 s, with improvement of 23%. However, *GRID-GUM*1.1 shows less improvement in case 10. It decreases the run-time required to finish the computation in two PEs, one local PE with slow CPU speed and one remote PE with fast CPU speed ($1E1M$), from 821 s to 699 s with improvement of 14%.

The behaviour of `queens` under *GRID-GUM*1.1 is mainly affected by the number of PE(s) with fast CPU speed which participate in the computation.

*GRID-GUM*1.1 always favours PEs with fast CPU speed to start and finish the computation, even if the PE with fast CPU speed is a remote PE. This could work effectively with `queens`, due to `queens`' communication and parallelism regularity. `queens`, as explained before, has a relatively low-communication degree. Due to this, the high-latency between PEs has a marginal impact on the performance. In addition, `queens`' regular parallelism enables *GRID-GUM*1.1 to show improvement in all cases.

As depicted in the seventh column in Table 5.17, the dynamic information of loads and latencies shows modest improvement on the performance.

For instance in case 2, to finish the computation of `queens` in five PEs ($2E3M$), two local PEs with slow CPU speed and three remote PEs with faster CPU speed, *GRID-GUM*1.1 requires 412 s. Nevertheless, under *GRID-GUM*2 `queens` requires 346 s with improvement of 12% in comparison to the performance under *GRID-GUM*1.1. In this case, *GRID-GUM*2 assists the two E PEs with slow CPU speed to get enough work, whose are neglected under *GRID-GUM*1.1. However, this improvement is depressed when fewer PEs with fast CPU speed participate in the computation.

For instance in case 9, to finish the computation of `queens` in three PEs ($2E1M$), two local PEs with slow CPU speed and one PE with faster CPU speed, under *GRID-GUM*1.1 requires 659 s. *GRID-GUM*2 supports `queens` to finish the computation on 639 s with a very modest improvement of 3% compared to *GRID-GUM*1.1. The dynamic information in *GRID-GUM*2 assists PEs to send the FISH messages to the PE with enough work to contribute to the performance with preference to PEs connected over low-latency communication. Moreover, the dynamic information supports *GRID-GUM*2 to distribute the work between PEs according to the amount of work available and the PE's CPU speed. However, `queens` generates relatively few tasks, moreover, these tasks likely to have the same weight of computation. All these explain the modest impact of the dynamic information in *GRID-GUM*2 on the performance.

In brief, Table 5.17 shows that, the static and dynamic information of the

adaptive load distribution of *GRID-GUM2* has contributed to improve the performance of `queens` in multi heterogeneous clusters on high-latency communication

.

Summary

- *GRID-GUM2* outperforms *GRID-GUM1* on heterogeneous high-latency architecture on our test programs (Tables 5.15, 5.16 and 5.17).
- For `raytracer` which has relatively high-communication degree and high irregular parallelism, *GRID-GUM2* achieves a substantial improvement of up to 60% under heterogeneous high-latency architecture (Table 5.15).
- Under the same architecture, for both programs with relatively low-communication degree (`sumEuler` and `queens`), *GRID-GUM2* achieves modest improvement of up to 31% and 35% with `sumEuler` and `queens` respectively (Tables 5.16 and 5.17).
- *GRID-GUM2* achieves less variability in the improvement with `queens`, with low-communication degree and regular parallelism, compare with `raytracer`, with high-communication degree and irregular parallelism. On heterogeneous high-latency architecture `queens`'s performance is improved between 19% and 35%, however, `raytracer`'s performance is improved between 4% and 60% (Tables 5.17 and 5.15).

5.4.2 High-Latency: Homogeneous Performance

This experiment investigates the performance impact of the adaptive load distribution of *GRID-GUM2* on high-latency homogeneous architecture.

This experiment does not include measurements from the *GRID-GUM1.1* implementation, due to the similarity in the performance under *GRID-GUM1*. Static information on performance has little impact on the homogeneous architecture used in this experiment. Thus, most of the improvements gain under this

architecture are from the dynamic information of loads and latencies.

The measurements in Tables 5.18 (`raytracer`), 5.19 (`sumEuler`) and 5.20 (`queens`) are performed on two homogeneous Beowulf clusters connected with high-latency communication, the Muni and Edin2 Beowulf clusters described in Tables 5.9 and 5.10. Each Edin2 machine is labelled E and each Muni machine is labelled M.

raytracer

In Table 5.18, the first and second columns show case number and different combinations of PEs. The third and fourth columns record the run-time in seconds for *GRID-GUM*1 and *GRID-GUM*2 respectively. The last column shows the percentage improvement in *GRID-GUM*2 run-time.

Case		Run-time s		Impr%
		*GRID-GUM*1	*GRID-GUM*2	
1	1E4M	995	617	38%
2	1E3M	1066	728	32%
3	2E3M	911	703	23%
4	1E2M	1088	892	18%
5	2E2M	952	843	11%
6	2E1M	1007	926	8%
7	1E1M	1842	1687	8%
8	3E1M	852	786	8%
9	4E1M	668	642	4%
10	3E2M	772	754	2%

Table 5.18: `raytracer`: Homogeneous High-Latency Computational Grid

In Table 5.18, *GRID-GUM*2 shows improvement in the run-time under different configurations of local and remote PEs in comparison with *GRID-GUM*1's performance, for instance, in case 2 `raytracer` on *GRID-GUM*1 using four PEs ($1E3M$), one local PE from Edin2 cluster and three remote PEs from Muni cluster, requires 1066 s. However, `raytracer` on *GRID-GUM*2 using the same

configuration ($1E3M$) requires only 728 s with an improvement of 32% in the run-time. Furthermore, *GRID-GUM2* in a configuration of the form $xEyM$, where $x < y$, has the highest improvement against *GRID-GUM1*. This is because in *GRID-GUM1*, the main PE is always selected as the first PE in the list, which has to be a E PE in this case. This suggests that the remote M PEs have to communicate through a high-latency communication with the main PE, which has all the work at the beginning of the execution, to obtain work, which becomes a bottleneck in the execution. However, *GRID-GUM2* prevents this bottleneck and decreases communication through high-latency as will be explained next. The adaptive load distribution of *GRID-GUM2* in a configuration of the form $xEyM$, where $x < y$, nominates the main PE from the M remote group PEs. By this nomination, the number of PE(s) required to obtain work through the high-latency communication decreases. Moreover, in *GRID-GUM2* for the same configuration, if a E PE sends a FISH message to a M PE, the M PE grants the E PE with enough work, depending on the amount of work available in M and E PEs and the number of M PEs (y) and E PEs (x) (as explained in Section 4.5), to decrease the amount of communication through high-latency communication.

In short, Table 5.18 shows that, *GRID-GUM2* outperforms *GRID-GUM1* on high-latency homogeneous architecture for `raytracer`, with high-communication degree and high irregular parallelism.

In Tables 5.19 (`sumEuler`) and 5.20 (`queens`) the first column shows different combinations of PEs. The second and third columns record the run-time in seconds for *GRID-GUM1* and *GRID-GUM2* respectively. The last column shows the percentage improvement in *GRID-GUM2* run-time. In this experiment `queens` and `sumEuler` show modest improvements under *GRID-GUM2*. This is due to the nature of these programs. `sumEuler` and `queens` have relatively low-communication degree, thus, the impact of high-latency communication is limited.

sumEuler

In Table 5.19, *GRID-GUM2* shows more improvements against *GRID-GUM1* with `sumEuler` than with `queens` in Table 5.20. This is due to the parallelism regularity in `sumEuler` and `queens`. `sumEuler` is based on a highly irregular parallelism which increases the need of an adaptive load distribution on a homogeneous high-latency architecture more than for `queens` which is based on regular parallelism.

Case		Run-time s		Impr%
		GRID-GUM1	*GRID-GUM2*	
1	2E2M	182	128	30%
2	3E2M	179	127	29%
3	2E3M	163	126	23%
4	2E1M	204	165	19%
5	1E4M	138	112	19%
6	4E1M	175	143	18%
7	1E3M	141	118	16%
8	3E1M	158	145	8%
9	1E1M	212	200	6%
10	1E2M	151	144	5%

Table 5.19: `sumEuler`: Homogeneous High-Latency Computational Grid

In Table 5.19, *GRID-GUM2* outperforms *GRID-GUM1*. For instance, in case 1, *GRID-GUM1* requires 182 s to finish the computation of `sumEuler` using four PEs ($2E2M$), two local E PEs and two remote M PEs. However, *GRID-GUM2* requires only 128 s to finish the same computation using the same configuration ($2E2M$) with improvement of 30%. In this case, *GRID-GUM2* chooses PEs to use the available dynamic information about loads and latencies to obtain work from PEs are known to be heavily loaded with preference to PEs known to have low-communication latency. Due to this, PEs spend less time seeking work and send less messages through the high-latency communication which result in the improvement of the performance on homogeneous architecture even if the program has relatively low-communication degree like `sumEuler`.

In cases 9 and 10 `sumEuler` shows modest improvement under *GRID-GUM2* of 6% and 5% respectively. This due to the combination of local E PE(s) and remote M PE(s) which participate in the computation. The lazy mechanism is used in *GRID-GUM2* to distribute dynamic information is less effective in a small set of PEs with relatively low-communication degree programs. This mechanism in a small set of combination of local and remote PEs might supply a PE with an initial dynamic information which shows that other PEs do not have enough work to share with the current PE. Thus, a PE with this incomplete and out-of-date dynamic information might finish its computation earlier than the rest of the PEs and stop seeking work, considering that other PEs do not have work to share.

In short, regarding the characteristic of the low-communication degree in `sumEuler`, *GRID-GUM2* shows a good improvement to the performance in comparison to *GRID-GUM1*. In addition, despite the lazy way of distributing the dynamic information under *GRID-GUM1*, the performance of `sumEuler` has not been impinge under any cases.

queens

Case		Run-time s		Impr%
		GRID-GUM1	*GRID-GUM2*	
1	3E1M	365	333	9%
2	4E1M	319	295	8%
3	1E1M	521	484	7%
4	2E1M	388	370	5%
5	3E2M	272	258	5%
6	1E2M	375	356	5%
7	2E2M	314	298	5%
8	2E3M	276	269	2%
9	1E3M	302	296	2%
10	1E4M	261	263	0%

Table 5.20: `queens`: Homogeneous High-Latency Computational Grid

In Table 5.20, *GRID-GUM2* scores modest improvements against *GRID-GUM1* with `queens`. The improvements lay between 9% and 0%. The number of local E PEs and remote M PEs effect directly the improvement. For instance in cases 1 and 2 where the number of local E PEs are more than the number of M PE(s), `queens` scores better improvement under *GRID-GUM2* compared with the cases 8, 9 and 10 where the number of remote M PEs is more than local E PE(s). This is because the remote M PEs are slightly faster (1529 MHz) than the local E PEs (1395 MHz). Thus, in the cases where the number of M PEs is more than the number of E PEs the adaptive load distribution of *GRID-GUM2* nominates the mainPE from the remote M PEs . This nomination decreases the improvement due to the possible delay in the startup. If the mainPE is nominated from the remote M PEs, the local E PE(s) which participate in the computation has to wait until the remote mainPE starts the computation. In a high-latency architecture this delay is remarkable especially in a program like `queens` which has regular parallelism and relatively low-communication degree, which does not leave a much possible improvement to *GRID-GUM2*.

Broadly speaking, in a homogeneous computational Grid architecture, a blind load distribution mechanism, similar to the load distribution in *GRID-GUM1*, delivers an improvement in the performance of programs with low-communication degree and regular parallelism like `queens`.

Summary

- The measurements in this section suggest that *GRID-GUM2* outperforms *GRID-GUM1* on homogeneous high-latency computational Grid architectures on our example programs;

- The performance of `raytracer`, with high-communication degree, shows an improvement of up to 37% using *GRID-GUM2* on homogeneous high-latency architectures (Table 5.18);

- Under the same architectures, the performance of `sumEuler`, with low-communication degree and irregular parallelism, shows an improvement of

up to 30% using *GRID-GUM2* (Table 5.19);

- However, *GRID-GUM2* scores the lowest improvement of up to 9% with `queens`, which has low-communication degree and regular parallelism, on homogeneous high-latency architectures; due to the little impact of this architecture in `queens`'s performance, which leaves little possibility for improvement, (Table 5.20).

5.5 Scalability

This section studies the scalability of the *GRID-GUM2* load distribution mechanism on heterogeneous architecture over high-latency communication. This experiment has the following limitations:

- Most of the functional parallel algorithms which are available, including the ones have been used in this thesis, do not scale to a large GRID environment, due to the lack of parallelism generated during the execution. These programs were originally implemented to work on conventional high performance computers which were built around a limited number of PEs. This limitation can be recognised from the `raytracer`'s results in this section.

- The number of PEs were available to be used in this experiment was limited by the number of PEs available in the cooperated sites, Edin1 (E) 30, Edin2 (E_2) 5, and Muni (M) 6.

In Tables 5.21 (`raytracer`), 5.23 (`parFib`) and 5.22 (`raytracer`), the measurements are performed in three heterogeneous Beowulf clusters, Edin1 and Edin2, which are connected over a low-latency communication, and Muni, which is connected with the other two clusters over a high-latency communication. For more details see Tables 5.9 and 5.10. Each Edin1 machine is labelled E, each Edin2 machine is labelled E_2 and each Muni machine is labelled M.

In Table 5.21 the first column shows case number. The second and fifth columns present number of PEs from Edin1 Beowulf cluster used with GUM and

different combinations of PEs from Edin1, Edin2 and Muni Beowulf clusters used with *GRID-GUM*1 respectively. The third and sixth columns record runtime in seconds under GUM and *GRID-GUM*1 respectively. The fourth and last columns indicate the speedup under GUM and *GRID-GUM*1.

In Tables 5.22 and 5.23, the first and second columns show case number and different combinations of PEs from Edin1, Edin2 and Muni Beowulf clusters. The third and fifth columns record the run-time in seconds for *GRID-GUM*1 and *GRID-GUM*2 respectively. The fourth and last columns indicate the speedup.

In these measurements the input size used with `raytracer` and `parFib` is relatively large, due to this, it was very hard to run these programs sequentially to calculate the speedup. Hence, the sequential run for `raytracer` is computed from the run-time of $7E$ from Table 5.21 under GUM and for `parFib` is computed from the runtime of $6E1M$ from Table 5.23 under *GRID-GUM*1.

5.5.1 raytracer

Case	raytracer					
	GUM			*GRID-GUM*1		
		Rtime s	Speedup		Rtime s	Speedup
1	7E	2609	7	6E1M	2530	7
2	14E	2168	8	12E2M	2185	8
3	21E	1860	10	18E3M	1824	10
4	28E	1771	10	24E4M	1776	10
5	30E	1762	10			
6				30E5M	1666	11
7				$5E_2$30E6M	1652	11

Table 5.21: `raytracer`: Scalability in GUM and *GRID-GUM*1

The `raytracer` performance in Table 5.21 shows that replacing local E PE with remote M PE, using high-latency communication, does not deteriorate the performance. In the first four cases, GUM and *GRID-GUM*1 score the same speedup, although, case 1 and 3 show better performance under *GRID-GUM*1

due to the fast M PEs participating in the computation. In contrast, case 2 and 4 show better performance under GUM in comparison to *GRID-GUM*1 due to the low-latency communication between PEs and the small time required for startup. However, GUM scalability is limited within a single cluster. As shown in the Table, GUM can only scale up to thirty PEs which is the size of Edin1 Beowulf cluster, which limited the improvement. In contrast *GRID-GUM*1 can scale with cluster size and uses PEs from other clusters to improve the performance as shown in case 6 and 7. These results support the idea of using GRID-emergent technology to improve the performance of parallel programs.

Case		raytracer			
		*GRID-GUM*1	Speedup	*GRID-GUM*2	Speedup
1	6E1M	2530	7	2470	7
2	12E2M	2185	8	1752	10
3	18E3M	1824	10	1527	12
4	24E4M	1776	10	1359	13
5	30E5M	1666	11	1278	14
6	$5E_2$30E6M	1652	11	1133	16

Table 5.22: `raytracer`: Scalability in *GRID-GUM*1 and *GRID-GUM*2

In Table 5.22, *GRID-GUM*2 outperforms *GRID-GUM*1 under different configurations of heterogeneous PEs using relatively a large set number of PEs with `raytracer` to calculate an image with resolution $800X800$. For instance in case 6, `raytracer` on *GRID-GUM*1 using 41 PEs ($5E_2$30E6M), five PEs from Edin2, thirty PEs from Edin1 and six PEs from Muni Beowulf clusters, scores a speedup of 11, however, *GRID-GUM*2 increases the speedup of `raytracer` under the same configuration to 16. This improvement shows the capabilities of the adaptive load distribution of *GRID-GUM*2 to improve the performance in a large set of PEs.

Figures 5.28, and 5.29 depict the per-PE profile of *GRID-GUM*1 and *GRID-GUM*2 performance presented in case 6 in Table 5.22 respectively.

In Figure 5.28, PEs have numerous idle periods, and finish at different time. In addition, all PEs except PE0 stop at 30% of the execution time. As explained

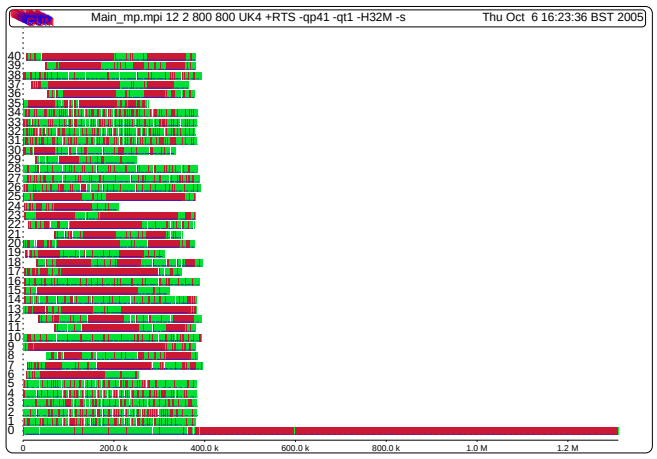


Figure 5.28: *GRID-GUM*1: per-PE activity profile for `raytracer` on 41 PE

before, this behaviour is due to the blind load distribution of *GRID-GUM*1. PEs under *GRID-GUM*1 send FISH messages to random destinations. There is, therefore no guarantee that a FISH message will be sent to a PE with work to donate in preference to a PE with no work; in fact there is no guarantee that the message will reach an arbitrary PE at all. As in the case with a relatively large number of PEs, the probability is higher, and it might be the case, that the FISH message bounces back and forth between a small group of PEs which do not have work to donate.

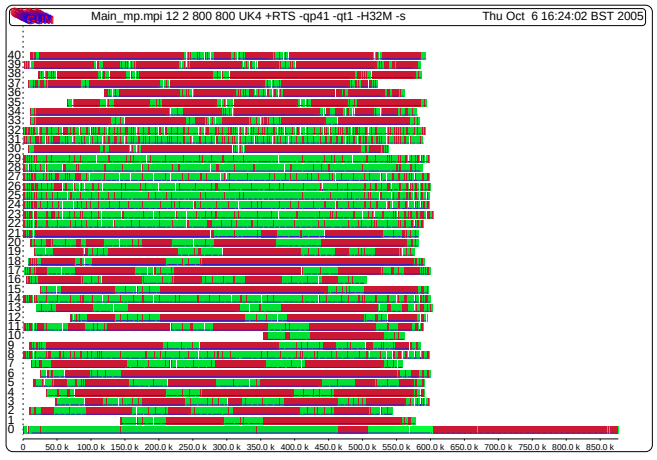


Figure 5.29: *GRID-GUM*2: per-PE activity profile for `raytracer` on 41 PE

In Figure 5.29, PEs only stop after almost 70% of the execution time. However, PEs still have some idle periods, is due to a combination of PEs and program characteristics. In a multi-cluster GRID environment, especially with a relatively large set of PEs, latency causes significantly different startup times and delays in the initial work request of the individual PEs. In addition, the `raytracer` program generates all parallelism at the beginning of the computation. Each of the parallel threads needs the input data early on in the computation. Therefore the thread will block almost immediately, fetching remote data, and the now idle PE will ask for additional work. *GRID-GUM2* controls the load distribution by encouraging PEs with slow CPU speed to be less aggressive for accumulating sparks. This avoids PEs with slow CPU speed from activating a huge number of sparks. However, it potentially increases idleness among PEs with slow CPU speed. In the execution shown in Figure 5.29 the 5 Edin2 PEs ($PE0$, $PE22$, $PE23$, $PE24$, $PE25$) and 6 Muni PEs ($PE26$, $PE27$, $PE28$, $PE29$, $PE31$, $PE32$) obtain the available parallelism and show a fairly balanced load and finish at about the same time. The other PEs with slow CPU speed remain with some idle periods, and are unable to extract extra work from PEs with fast CPU speed until they finish their current computation. Moreover, at the end of the computation, all PEs require data initially held on the mainPE without having sparks available, which explain the tail on PE0 at the end of the execution. Despite all these *GRID-GUM2* improves the performance by 31%.

Figure 5.30 plots the speedup of `raytracer` under GUM, *GRID-GUM1* and *GRID-GUM2*. As depicted in the figure, GUM shows limited capability to scale using as many PEs as *GRID-GUM1* and *GRID-GUM2*. *GRID-GUM1*'s load distribution is incapable to use the available computation power resources to improve the parallel program performance, in contrast to *GRID-GUM2*.

5.5.2 parFib

Table 5.23 presents the performance of `parFib` under *GRID-GUM1* and *GRID-GUM2*. `parFib` has a relatively low-communication degree compare to `raytracer`,

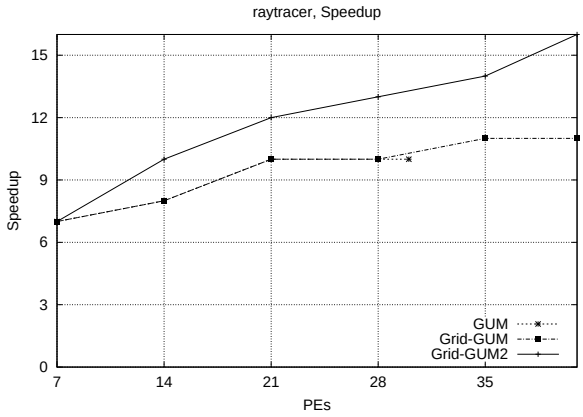


Figure 5.30: `raytracer`: GUM, *GRID-GUM*1 & *GRID-GUM*2

it sends approximately 14 Msgs/s (Table 5.14). Moreover, `parFib` has highly regular parallelism and it generates packets with small size (5.8 Byte). In fact, `parFib` is a perfect program which can manage heterogeneity and high-latency in computational GRID architectures without relying on an adaptive load distribution. Due to this, `parFib` shows a good performance under *GRID-GUM*1's blind load distribution.

This experiment explores the impact of using *GRID-GUM*2's adaptive load distribution with a perfect program like `parFib` in heterogeneous computational GRID architecture over high-latency communication.

Case		`parFib`				Impr%
		*GRID-GUM*1	Speedup	*GRID-GUM*2	Speedup	
1	6E1M	3995	7	3737	7	0%
2	12E2M	1993	14	2003	14	0%
3	18E3M	1545	18	1494	19	5%
4	24E4M	1237	23	1276	22	-4%
5	30E5M	1142	24	1147	24	0%
6	$5E_2$30E6M	1040	27	1004	28	4%

Table 5.23: `parFib`: Scalability in *GRID-GUM*1 and *GRID-GUM*2

In Table 5.23, `parFib` depicts the same performance under *GRID-GUM*1 and *GRID-GUM*2. Case 3, 4, and 6 show marginal differences in the performance

within the range of −4% and +5%, due to the impact of operating system and shared communication interaction.

The measurements in Table 5.23 shows the effectiveness of the light-weight implementation of the monitoring mechanism in *GRID-GUM2*. It also shows that the adaptive load distribution of *GRID-GUM2* does not have a negative impact on the performance of a program which can independently score a good performance in computational Grid architecture.

5.5.3 Summary

- The experiment in this section exhibits that, emerging Grid technology offers the opportunity to improve performance by integrating PEs from different clusters located in different geographical places into a computational Grid architecture (Table 5.21 and Figure 5.30)
- The measurements in the section show that, *GRID-GUM2* improves the performance of `raytracer` in a relatively large scale set of PEs (Table 5.22)
- The experiment of running `parFib` in a large scale architecture suggests that *GRID-GUM2* has a light-weight monitoring implementation and does not effect the performance of (Table 5.23).

5.6 Conclusion

This chapter includes systematic measurements of the performance of GpH parallel programs using the adaptive load distribution of *GRID-GUM2*, on a range of computational Grid architectures. We conclude that the new adaptive load distribution of *GRID-GUM2* improves parallel performance for a typical set of applications running on homogeneous and heterogeneous computational Grids architectures over high and low-latency communications. However, the adaptive load distribution of *GRID-GUM2* has different impact in the performance for different configurations of latency and heterogeneity, and programs characteristics, i.e. communication degree and regularity of parallelism.

Broadly speaking, we conclude that, with appropriate load management strategies, acceptable performance can be obtained on computational GRID architectures from a distributed virtual shared heap implementation of a high-level parallel language, as shown in Table 5.24.

Table 5.24 compares the performance of our benchmark programs using *GRID-GUM*1, *GRID-GUM*1.1 and *GRID-GUM*2. The first column shows the program name. The second and third columns present the program characteristics, parallelism regularity and communication degree, respectively. The fourth and fifth columns include the framework test environment specification, latency and architecture heterogeneity or homogeneity. The sixth, seventh and eighth columns report the performance of *GRID-GUM*1 (*GG*1), *GRID-GUM*1.1 (*GG*1.1) and *GRID-GUM*2 (*GG*2). In this comparison, we rate our performance from 3 (best) to 1 (worst).

We can conclude from Table 5.24:

- *GRID-GUM*2 improves the performance of the benchmark programs with irregular parallelism on heterogeneous architecture.
- *GRID-GUM*2 improves the performance of all benchmark programs except the programs with regular parallelism and low-communication degree on high-latency and homogeneous computational GRIDs.
- *GRID-GUM*2 improves the performance of the benchmark programs with irregular parallelism on low-latency and homogeneous computational GRIDs and sustain the same performance with the benchmark programs with regular parallelism.
- *GRID-GUM*2 shows the same improvement as *GRID-GUM*1.1 with regular parallelism program with heterogeneous architecture.
- *GRID-GUM*1.1 improves the performance of the benchmark programs with irregular parallelism with heterogeneous architecture.

Program	Characteristic		Framework		GG1	GG1.1	GG2
	regularity	Comm.	Latency	Archit.			
raytracer	Irreg.	High	High	Hetr.	1	2	3
			High	Hom.	1	1	2
			Low	Hetr.	1	2	3
			Low	Hom.	1	1	2
sumEuler	Irreg.	Low	High	Hetr.	1	2	3
			High	Hom.	1	1	2
			Low	Hetr.	1	2	3
			Low	Hom.	1	1	2
linSolv	Irreg.	Low	High	Hetr.	1	2	3
			High	Hom.	1	1	2
			Low	Hetr.	1	2	3
			Low	Hom.	1	1	2
matMult	Reg.	High	High	Hetr.	1	2	2
			High	Hom.	1	1	2
			Low	Hetr.	1	2	2
			Low	Hom.	1	1	1
queens	Reg.	Low	High	Hetr.	1	2	2
			High	Hom.	1	1	1
			Low	Hetr.	1	2	2
			Low	Hom.	1	1	1
parFib	Reg.	Low	High	Hetr.	1	2	2
			High	Hom.	1	1	1
			Low	Hetr.	1	2	2
			Low	Hom.	1	1	1

Table 5.24: Summary comparison between *GRID-GUM*1, *GRID-GUM*1.1 and *GRID-GUM*2

More specifically, Tables 5.15 and 5.18 show that the adaptive load distribution of *GRID-GUM*2 can give significant improvements with `raytracer`, which has high-communication degree and high irregular parallelism, on high-latency heterogeneous and homogeneous computational GRIDs. This improvement is modest with `sumEuler`, which has relatively low-communication degree and irregular parallelism, as depicted in Tables 5.16 and 5.19. Finally, `queens`, which has relatively low-communication degree and regular parallelism, shows less improvement under *GRID-GUM*2 for high-latency communication under homogeneous

and heterogeneous architecture as shown in Tables 5.20 and 5.17. The performance of the programs with relatively low-communication degree, like `sumEuler` and `queens`, are less effected by the high-latency communication. Due to this, the possible improvement that can be achieved by the adaptive load distribution of *GRID-GUM2* is limited. Moreover, the improvement possibilities decrease when the program is based on regular parallelism, like `queens`.

However, the adaptive load distribution of *GRID-GUM2* improves the performance significantly on low-latency heterogeneous computational GRIDs. Most of the programs exhibit an improvement of over 50%, (`raytracer` 57%, `queens` 53%, and `sumEuler` 51%), as depicted in Table 5.12.

Moreover, as shown in Table 5.14 on low-latency homogeneous computational GRIDs, the monitoring and adaptive load distribution of *GRID-GUM2* retains a very light overhead. Although *GRID-GUM2* reduces the communication degree other dynamic properties of the programs remain unchanged. The communication degree decreases under *GRID-GUM2* due to the dynamic and static information are supplied by the monitoring mechanism in *GRID-GUM2*, which assist PEs to seek work in the right place, thus the communication degrees are decreased.

Finally, the adaptive load distribution of *GRID-GUM2* reduces all programs run-time. This is due to the accuracy of the monitored dynamic and static information which assist PEs to locate work from heavily loaded PEs. Thus, the adaptive load distribution of *GRID-GUM2* delivers more predictable performance in comparison to the load distribution of *GRID-GUM1* as shown in Table 5.13.

Chapter 6

Conclusion

6.1 Contributions

Computational GRIDs are an emerging technology that manage and integrate resources and services distributed across multiple control domains. Computational GRIDs are comprised of large-scale networks of computers and many offer huge computational power, but currently suffer from a lack of automatic management of parallelism. In this thesis we have shown that with dynamic, implicit task management it becomes possible to exploit complex parallelism within applications and thus to speed-up their execution. This approach differs from many of today's GRID applications which are largely restricted to parallelising independent jobs , executing the same computation on different data items.

More specifically, this thesis has used GPH, a language with high-level parallel coordination that abstracts over the complexities of the architecture. Based on the GUM runtime environment for GPH we have developed a novel runtime environment for computational GRIDs, *GRID-GUM2*, that automatically monitors and manages the load distribution on computational GRIDs.

6.1.1 Contribution 1: GUM on the GRID

The thesis first modified the GUM *runtime environment for deployment over a wide area network, realising GRID-GUM1* [AZTML03]. *In addition, the thesis*

reports a systematic series of GRID-GUM1 performance measurements on different configurations of computational GRIDs *with widely varying latencies and programs with different communication degree. It also compares the performance under PVM and MPICH communication libraries with the* GRID *implementation of the MPI standardMPICH-G2* [AZTLM04].

In developing *GRID-GUM2*, a crucial first step was to port GUM to the Globus GRID implementation. In particular, it was important to demonstrate conclusively that the HPC-oriented GUM communication layer could be adapted for computational GRIDs. GUM's communication layer is based on PVM, however, the Globus Toolkit communication is based on a special form of MPI, MPICH-G2. This led to change the communication layer in GUM as a first step towards implementing *GRID-GUM1*. (Section 3.2)

There have been few systematic measurements of high-level parallel languages on computational GRIDs [Con04, Mur03, GAL+02]. The measurements of *GRID-GUM1* show that the performance of substantial programs on a single cluster is independent of the underlying communication library. On multiple GRID-enabled clusters with a low latency interconnect, *GRID-GUM1* delivers good and predictable speedups.

In contrast, on clusters with a high latency interconnect *GRID-GUM1* only delivers acceptable speedups for programs that perform little communication. Poor load balance is identified as the performance bottleneck (Section 3.3).

We conclude that to give good performance for a high level language on a shared hierarchical heterogeneous architecture like a GRID, more elaborate RTE techniques are required to control work distribution and distributed memory management (Section 3.4).

6.1.2 Contribution 2: Design of an Adaptive RTE for Computational GRIDs

The thesis presents the design of the GRID-GUM2 RTE with novel load balancing mechanisms for shared hierarchical heterogeneous architectures. GRID-GUM2 is the first fully implemented virtual shared memory RTE that dynamically manages parallel execution on computational GRIDs [AZTLM05, AZMLT04].

*GRID-GUM*2 extends *GRID-GUM*1's simple dynamic adaptive mechanisms to reflect both differences in the underlying process elements and network characteristics, and dynamic changes in external processing and communication loads in computational GRIDs (Section 4.3).

*GRID-GUM*2, unlike traditional high performance computing environments, automatically manages key coordination aspects such as task placement and communication and synchronisation within a program in computational GRIDs. The management policies of *GRID-GUM*2 is informed by the static properties (Section 4.4.1) e.g. moving more work to larger or faster cluster and dynamic properties, (Section 4.4.2) e.g. lightly-loaded clusters become aggressive about seeking work. *GRID-GUM*2 is extended to manage the hierarchical architecture, e.g. seeking work within a cluster before looking outside. The mechanisms are decentralised, obtain complete static information during start up, and then cheaply propagating partial dynamic information during execution (Section 4.5).

6.1.3 Contribution 3: Evaluating *GRID-GUM*2's Dynamic Load Scheduling on Computational GRIDs

This thesis demonstrates that lightweight adaptive load distribution techniques, like those in GRID-GUM2, can deliver good performance for a typical set of applications on both high- and low- latency, and both homogeneous and heterogeneous computational GRIDs [AZTLM06].

Systematic measurements have been conducted into the performance of GpH parallel programs, which do not contain code for explicit coordination of parallelism, using the automatic and adaptive load distribution of *GRID-GUM2*, on a range of computational Grids.

Broadly speaking, the experiments presented in Chapter 5 show that *GRID-GUM2*'s new load distribution mechanism improves the performance of all presented programs in this thesis, as shown in Table 5.24.

More specifically, the experiments suggest that, on high-latency networks in a heterogeneous and homogeneous architecture, the adaptive load distribution of *GRID-GUM2* (Section 5.4):

- shows significant improvements for program with high-communication degree and high irregular parallelism (e.g. `raytracer`);
- gives modest improvement for programs with relatively low-communication degree and irregular parallelism (e.g. `sumEuler`);
- shows little improvement for programs with relatively low-communication degree and regular parallelism (e.g. `queens`).

However, on the low-latency network in a heterogeneous architecture the adaptive load distribution of *GRID-GUM2* improves the performance significantly, with some programs exhibiting an improvement of over 50%, (`raytracer` 57%, `queens` 53%, and `sumEuler` 51%) (Section 5.3.1).

Moreover, on the low-latency in a homogeneous architecture, programs under *GRID-GUM2* have less variance in the performance than under *GRID-GUM1*. *GRID-GUM2* shows good scalability and retains a very light overhead which does not effect the program's dynamic properties (Section 5.3.2).

In short, this thesis concludes that, with appropriate load management strategies, acceptable performance can be obtained on computational Grids *from a distributed virtual shared heap implementation of a high-level parallel language without explicit constructs for coordinating the parallel evaluation.*

6.2 Limitations & Future Research Direction

The current work has the following limitations,

- All the programs measured in this thesis can be classified as medium scale parallel programs.
- The security mechanism used in this thesis is inherited from Globus Toolkit. Moreover, it focuses on user authentication and is based on trust towards the user. There are no program-based security mechanism.
- The implementation presented in this thesis, (*GRID-GUM2*), uses the communication library MPICH-G2 because of the support and services from the Globus Toolkit. However, MPICH-G2 suffers from being a closed system, which means it is impossible to introduce a new PE to the computation after it started. Due to this, *GRID-GUM2* restricted to be a closed system.
- This thesis does not introduce any fault tolerant mechanism. If a PE fails for some reason then the entire computation may fail.

There are several avenues to extend this research and address the limitations has been identified.

Larger Algorithms: The first avenue is to implement a large parallel algorithm for more experiments to show scalability in computational GRIDs. An ongoing project (*SCIEnce - Symbolic Computation Infrastructure for Europe EU FP VI I3-026133*) will start on April/2006 and will integrate symbolic computation tools with computational GRIDs using *GRID-GUM2* to tackle new problems that cut across the traditional divisions between system specialities. Symbolic computer algorithms are likely to exhibit characteristics that fit well to *GRID-GUM2* (highly irregular).

Program-Based Security: Another possible avenue is to implement a program based security mechanism to analyse program behaviour to decide whether

to permit execution of the code. For example, to enhanced program based security, certificates of bounded resource consumption should be attached to the code sent between PEs in the network and checked by a resource protection component before executing the code.

Open System The third avenue to be explored is to target different communication libraries than MPICH-G2, to enable *GRID-GUM2* to be an open system. Possibility includes using an optimised version of PVM for computational GRIDs [MS99, SK03].

Fault Tolerance Future research must tackle the problem of fault tolerance especially in computational GRIDs. *GRID-GUM2* could benefit from the side-effect freed nature of functional language. One reason is that large part of the program are pure, i.e. without side effects. Side effects amount to updating the global program state and their absence means that the damage by a failing computation is confined. Moreover, if an error is detected, pure computations can be automatically restarted without the danger of making multiple updates. A second potential benefit of high-level language technology is that fault tolerance is global property affecting all operations of the virtual machine underlying a language, and enforcing such property is easier with a high level virtual machine. To investigate these potential benefits *GRID-GUM2* could have a new runtime system level of fault tolerance [TPL00].

Appendix A

Measurement Framework

A.1 Hardware Apparatus

The measurements have been performed on five Beowulf clusters: three located at Heriot-Watt Riccarton campus (*Edin1, Edin2,* and *Edin3*), a cluster located at Ludwig-Maximilians University Munich (*Muni*), and a cluster located at Heriot-Watt boarder campus(*SBC*); see Tables 5.9 and 5.10 for the characteristic of these Beowulfs.

Beowulfs	CPU	Cache	Memory
	Speed MHz	kB	Total kB
Edin1	534	128	254856
Edin2	1395	256	191164
Edin3	1816	512	247816
Muni	1529	256	515500
SBC	933	256	110292

Table A.25: Characteristics of Beowulf Clusters

A.2 Software Apparatus

The programs measured in this experiment are classified by the communication degree, which is the number of messages the program sends per second, so we can study the impact of the latency of the network on program behaviour. Six

	Edin1	Edin2	Edin3	SBC	Muni
Edin1	0.20	0.27	0.35	2.03	35.8
Edin2	0.27	0.15	0.20	2.03	35.8
Edin3	0.35	0.20	0.20	2.03	35.8
SBC	2.03	2.03	2.03	0.15	32.8
Muni	35.8	35.8	35.8	32.8	0.13

Table A.26: Approximate latency between clusters (ms)

programs are measured in this experiment. Three have low communication degree, `parFib`, `queens` and `sumEuler`, and the other three have relatively high communication degree, `raytracer`, `matMult`, and `linSolv`.

`parFib`:

The `parFib` program computes the number of calls to the Fibonacci function. The granularity of `parFib` can be controlled by specifying the threshold for parallel invocation which help manage the computation size. `parFib` is a divide-conquer program with regular parallelism.

`sumEuler`:

The `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. It is data parallel and has a fairly cheap combination phase involving only a small amount of communication. `sumEuler` has a high irregular parallelism.

`queens`:

The `queens` program places queen chess pieces on the chess board so that they do not check each other. `queens` is a divide-conquer program with regular parallelism.

`raytracer`:

The `raytracer` program calculates a 2D image of a scene of 3D objects by tracing all rays in a window. In tracing a ray, the intersections with the objects are computed. When an intersection is found, the ray is reflected and the colour of the intersection point is computed based on the strength of the ray and on the texture of the object's material. `raytracer` is data parallel program with high irregular parallelism. `raytracer` generates most of the parallelism at the beginning of the execution.

`matMult`:

The `matMult` program multiples two matrices. Given two square matrices of arbitrary precision integers $A, B \in \mathbb{Z}^{n \times n}, n \in \mathbb{N}$ find their product, i.e. a matrix $C \in \mathbb{Z}^{n \times n}$ such that $C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$. `matMult` is a divide-conquer program with regular parallelism.

`linSolv`:

The `linSolv` program finds an exact solution of a linear system of equations of the form $Ax = b$ where $A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^{n}, n \in \mathbb{N}$. `linSolv` is a symbolic algebra problem with data parallel paradigm and limited irregular parallelism.

Appendix B

Activity Profiles

The aim of the activity profiles to summarise the activity of the machine during the computation in one graph. In order to give the programmer the possibility of examining the program execution in more detail.

B.1 Overall Activity Profile

The idea of the overall activity profile is to present a global picture of the computation. In particular, it shows the utilisation of the machine at each point. The overall activity profile separates the threads into five different classes:

- running threads, i.e. threads that are currently performing a reduction, which are shown as a green area in the graph,
- runnable threads, i.e. threads that could be executed but that have not found an idle PE, which are shown as an amber area in the graph,
- blocked threads, i.e. threads that wait for a result that is being computed by another thread, which are shown as a red area in the graph.
- fetching threads, i.e. threads that are currently fetching data from a remote PE, which are shown as a light blue area in the graph,
- migration threads, i.e. threads that are currently being transferred from a busy PE to an idle PE, which are shown as a dark blue area in the graph.

The overall activity profile shows the number of threads (y-axis) in each class during execution time (x-axis).

B.2 Per-PE Activity Profile

The idea of the per-PE activity profile is to show the most important pieces of information about each PE in one graph. Therefor it is easy to compare the behaviour of the different PEs and to spot imbalances in the computation. This profile is often used to study runtime-system issues like the load balance in the system and is therefore most useful in a system-oriented view of parallelism. This profile can be regarded as a "load focusing" profile.

The per-PE activity profile shows one strip for each of the PEs. Each of the strips encodes three pieces of information:

- If the PE is active the strip appears in some shade of green, and if it is idle it appears in red.
- The load is measured by the number of runnable threads on the PE. A high load is shown by a dark shade of green.
- The number of blocked threads on the PE is shown by the thickness of a blue bar at the bottom of each strip.

Bibliography

[ABG02] M. Alt, H Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. In *CMPP'02 — Int. Workshop on Constructive Methods for Parallel Programming.* Dagstuhl, Berlin, June 2002.

[ABK+05] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, I. Dumitrescu, C.and Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In *High Performance Distributed Computing Conference (HPDC 14)*, November 2005.

[ADD04] M. Aldinucci, M. Dnelutto, and Dünnweber. Optimization Techniques for Implementing Parallel Sckeletons in Grid Environments. In *CMPP'04 — Intl. Workshop on Constructive Methods for Parallel Programming*, Stirling, Scotland, July 2004.

[ADF+01] G. Allen, T. Dramlitsch, I. Foster, N.T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 52–52, New York, NY, USA, 2001. ACM Press.

[AFFH98] E. Akarsu, G.C. Fox, W. Furmanski, and T. Haupt. WebFlow: high-level programming environment and visual authoring toolkit for high performance distributed computing. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–7, Washington, DC, USA, 1998. IEEE Computer Society.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AGG+99] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. McInnes, S. R. Parker, and B. A. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceeding of the 8th High Performance Distributed Computing (HPDC)*, 1999.

[AGK00] D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 520–528, Washington, DC, USA, 2000. IEEE Computer Society.

[AHS+] R. J. Allan, D. Hanlon, G. Smith, R. F. Fowler, and C. Greenough. A globus developers' guide with installation and maintenance hints. Technical report. Draft from 14 September 2001, <URL:`http://esc.dl.ac.uk/StarterKit/PS/globus_guide.ps/`>, fetch on May/2002.

[AJ89] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle v, G\rangle$-machine. In *FPCA'89 — Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, Imperial College, London, UK, September 1989. ACM Press.

[AJ90] L. Augustsson and T. Johnsson. *Lazy ML User's Manual*. System Manual, 1990.

[AS99] J. Almond and D. Snelling. UNICORE: uniform access to supercomputing as an element of electronic commerce. *Future General Computer System*, 15(5-6):539–548, 1999.

[AZMLT04] A. Al Zain, G. Michaelson, H-W. Loidl, and P. Trinder. Improving Load Balance in a Grid-enabled Parallel Haskell. In *TFP'04 — Intl. Workshop on Trends in Functional Programming*, Draft Proceedings, München, Germany, November 2004.

[AZTLM04] A. Al Zain, P. Trinder, H-W. Loidl, and G. Michaelson. Grid-GUM: Towards Grid-Enabled Haskell. In *IFL'04 — Intl. Workshop on the Implementation of Functional Languages*, Draft Proceedings, Lübeck, Germany, September 2004.

[AZTLM05] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell . In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005. to appear.

[AZTLM06] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *Journal of Scalable Computing: Practice and Experience*, 6(4), 2006.

[AZTML03] A. Al Zain, P. Trinder, G. Michaelson, and H-W. Loidl. Grid-GUM: Haskell on Grids. In *TFP'03 — Intl. Workshop on Trends in Functional Programming*, Draft Proceedings, Edinburgh, UK, September 2003.

[BA90] H. Ben-Ari, editor. *Principles of Concurrent and Distributed Programming*. Printce Hall, Englewood Cliffs, 1990.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *HPC ASIA'2000 China*. IEEE CS Press, USA, 2000.

[Bar84] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North Holland, 1984.

[BC97] L. Braine and C. Clack. An Object-Oriented Functional Approach to Information Systems Engineering. In *CaiSE'97 - Fourth Doctoral Consortium on Advanced Information Systems Engineering*, 1997.

[BDF+99] M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, M. Kohl, J.a nd Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam. HARNESS: A Next Generation Distributed Virtual Machine. *Future Generation Computer Systems*, 15(5/6):571–582, October 1999. Special Issue on Metacomputing.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

[Ber99] F. Berman. High-Performance Schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 12, pages 279–311. Morgan Kufmann, 1999.

[BFH03a] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. Chichester: John Wiley & Sons, 2003.

[BFH03b] F. Berman, G. Fox, and T. Hey. The Grid: past, present, future. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing - Making the Global Infrastructure a Reality*, pages 9–50, West Sussex, England, 2003. John Wiley & Sons, Ltd.

[BL99] J. Basney and M. Livny. *High Performance Cluster Computing*, volume 1, chapter Deploying a High Throughput Computing Cluster. Prentice-Hall, 1999.

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí. Eden — The Paradise of Functional Concurrent Programming. In *EuroPar'96 — European Conf. on Parallel Processing*, LNCS 1123, pages 710–713, Lyon, France, 1996. Springer.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Programming Models*, pages 120–124, Geneva, Switzerland, 1997. IEEE Computer Science Press.

[BRS94] F.W. Burton and V.J. Rayward-Smith. Worst Case Scheduling for Parallel Functional Programming. *Journal of Functional Programming*, 4(1):65–75, January 1994.

[BW97] F. Berman and R. Wolski. The AppLeS Project: A Status Report. Technical report, May 1997. 8th NEC Research Symposium, Berlin, Germany.

[CB97] C. Clack and L. Braine. Introducing CLOVER: an Object Oriented Functional Language. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, number 1268 in Lecture Notes in Computer Science. Springer Verlag, 1997.

[CD95] H. Casanova and J. Dongarra. Netsolv:a network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, 1995.

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[CFK99] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 37, Washington, DC, USA, 1999. IEEE Computer Society.

[CK88] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distribution Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[CK95] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distribution Computing Systems . In A. Shirazi, A. R. Hurson, and K. M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Transactions on Software Engineering, 1995.

[Cla99] C. Clack. Realisations for Non-Strict Langauges. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 6, pages 149–187. Springer-Verlag, 1999.

[Cli82] W. Clinger. Nondeterministic Call by Need is Neither Lazy Nor by Name. In *LFP'82 — Conf on LISP and Functional Programming*, pages 226–234, Pittsburgh, PA, 1982.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

[Col99] M. Cole. Algorithmic Skeletons. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 13, pages 289–304. Springer-Verlag, 1999.

[Com05] Platform Computing, 2005. <URL:`http://www.platform.com/`>.

[Con04] The ConCert Project. WWW page, 2004. `http://www-2.cs.cmu.edu/ concert/`.

[Con05] Condor, 2005. <URL:`http://www.cs.wisc.edu/condor/`>.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[CW88] R. G. Clark and L. B. Wilson. *Comparative Programming Languages*. Addison-Wesley, 1988.

[Dat05] DataGrid, 2005. <URL:`http://eu-datagrid.Web.cern.ch/`>.

[DB05] A. R. Du Bois. *Mobile Computation in a Purely Functional Language*. PhD thesis, School of Mathematic and Computer Science, Computer Science Department, Heriot-Watt University, Edinburgh, Scotland, 2005.

[DBLT02] A. R. Du Bois, H-W. Loidl, and P. Trinder. Thread Migration in a Parallel Graph Reducer. In *IFL'02 — Intl. Workshop on the Implementation of Functional Languages*. Springer-Verlag, LNCS 2670, 2002.

[Der02] J. Dermoudy. *Effective Runtime Management of Parallelism in a Functional Programming Context*. PhD thesis, University of Tasmania, 2002. Under assessment.

[DFF+02] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *The Sourcebook of Parallel Computing*. Morgan Kufmann, San Francisco, CA, USA, 2002.

[DFH+96] D. Diachin, L. Freitag, D. Heath, J. Herzog, W. Michels, and P. Plassmann. Remote engineering tools for the design of pollution control systems for commercial boilers. *International Journal of Supercomputer Applications*, 10(2):208–218, 1996.

[DFP+96] T. A. DeFanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide-Area Visual Supercomputing. *j-IJSAHPC*, 10(2/3):123–131, Summer/Fall 1996.

[DGT96] J. Darlington, Y. Guo, and H.W. To. Structured Parallel Programming: Theory meets Practice. In R. Milner and I. Wand, editors, *Research Directions in Computer Science*. Cambridge University Press, 1996.

[DR00] P. Drum and G Rackl. Applying and Monitoring Latency-Based Metacomputing Infrastructures . In *ICPP Workshops*, IEEE, pages 181–188, 2000.

[DRBJ02] D. De Roure, M. A. Baker, and N. R. Jennings. The Evolution of the Grid. *International Journal Computation and Currency: Practice and Experience*, June 2002.

[ELZ85] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract). In *SIGMETRICS '85: Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 1–3, New York, NY, USA, 1985. ACM Press.

[Esk89] M.R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems. In *IEEE Technical Comittee on Operating Systems Newsletter*, pages 3–13, Winter 1989.

[FAF05] FAFNER, 2005. <URL:`http://www.npac.syr.edu/factoring.html/`>.

[FGN+96] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high-performance distributed computing experiment. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*, pages 562–571, Washington, DC, USA, 1996. IEEE Computer Society.

[Fit01] S. Fitzgerald. Grid Information Services for Distributed Resource Sharing. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, pages 181–194, Washington, DC, USA, 2001. IEEE Computer Society.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[FK98a] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *SC'98: High Performance Networking and Computing: Proceedings of ACM IEEE SC98 Conference*, pages 7–13, Orange County Convention Center, Orlando, Florida, USA, November 1998.

[FK98b] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *HCW '98: Proceedings of the Seventh Heterogeneous Computing Workshop*, page 4. IEEE Computer Society, 1998.

[FK99a] I. Foster and C. Kesselman. Computational Grid. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, San Francisco, CA, USA, 1999. Morgan-Kaufman. Chapter 2.

[FK99b] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[FKNT02] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: Open Grid Services Architecture for Distributed Systems Integration. February 2002. published in

Open Grid Service Infrastructure WG, Global Grid Forum (GGF), <URL:`http:://www.globus.org/research/papers/ogsa.pdf`>.

[FKT96] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High performance computation Applications*, 2001.

[For05] The Global Grid Forum. , 2005. <URL:`http://www.gridforum.org/`>.

[Fos99] I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Globus Toolkit. Morgan-Kaufman, San Francisco, CA, USA, 1999.

[Fos02a] I. Foster. The Grid: A new infrastructure for 21st century science. In *Physics Today*, USA, 2002. American Institute of Physics.

[Fos02b] I. Foster. What is the Grid? A Three Point Checklist. *Daily news and information for the global grid community*, 1(6), July 22 2002. <URL:`http://www.gridtoday.com/02/0722/100136.html`>.

[FTL+01] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *HPDC10 — Tenth International Symposium on High Performance Distributed Computing*. IEEE Press, August 2001.

[FvL98] I. Foster and G. von Laszewski. Usage of LDAP in Globus. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[GAL+02] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *VECPAR'2002 — International Conference on Vector and Parallel Processing*, volume 2565 of *LNCS*, pages 197–227, Porto, Portugal, June 26-28, 2002. Springer-Verlag.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongerra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT, 1994.

[Geh84] N. Gehani. *Ada- Concurent Programming*. Prentice-Hall International, 1984.

[GKP96] G. A. Geist, J.A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Parallels*, 8(2), 1996.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[GLFK98] A.S. Grimshaw, M.J. Lewis, A.J. Ferrari, and J.F. Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems. Technical Report CS-98-12, Department of Computer Science, University of Virginia, 1998.

[Glo05] Globus, 2005. <URL:`http://www.globus.org/toolkit/`>.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT, second edition, 1999.

[Gnu05] Gnutella, 2005. <URL:`http://www.gnutella.co.uk/`>.

[Gos91] A. Goscinski, editor. *Distributed Operating System: The Logical Design*. Addison-Wesley, 1991.

[HAFF99] T. Haupt, E. Akarsu, G. Fox, and W. Furmanski. Web based metacomputing. *Future Gener. Comput. Syst.*, 15(5-6):735–743, 1999.

[HL00] C. Herrmann and C. Lengauer. $\mathcal{HDC}$: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — Conf. on High Performance Functional Computing*, pages 208–221, Denver, CO, Apr 10–12, 1995.

[HM99] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*, chapter Introduction. Springer-Verlag, 1999.

[HMP+95] K. Hammond, J.S. Mattson Jr., A.S Partridge, S.L. Peyton Jones, and P.W. Trinder. GUM: a Portable Parallel Implementation of Haskell. In *IFL'95 — Intl Workshop on the Parallel Implementation of Functional Languages*, September 1995.

[Hot05] HotPage, 2005. <URL:`http://www.hotpage.npaci.edu/`>.

[HP90] K. Hammond and S.L. Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *IFL'90 — Intl. Workshop on the Parallel Implementation of Functional Languages*, pages 51–72, Nijmegen, The Netherlands, June 1990.

[HP92] K. Hammond and S.L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *IFL'92 — Intl. . Workshop on the Parallel Implementation of Functional Languages*, pages 73–98, RWTH Aachen, Germany, September 1992.

[HPR00] F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *SFP'00 — Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, pages 11–19, St Andrews, Scotland, Jul 26–28, 2000. Intellect.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Hyd] P. Hyde. *Java Thread Programming*. Sams. ISBN 0-672-31585-8.

[JC01] M. E. Jerrell and W. A. Campione. The network-enabled optimization system (neos) - a means of solving optimization problems over the internet. Computing in Economics and Finance 2001 87, Society for Computational Economics, 2001. available at http://ideas.repec.org/p/sce/scecf1/87.html.

[JCS89] Simon L. Peyton Jones, Chris D. Clack, and Jon Salkild. High-performance parallel graph reduction. In Eddy Odijk, Martin Rem, and

Jean-Claude Syre, editors, *PARLE (1)*, volume 365 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 1989.

[Jin04] Jini, 2004. <URL:`http://www.jini.org/`>.

[JXT05] JXTA, 2005. <URL:`http://www.jxta.org/`>.

[KHT98] D.J. King, J. Hall, and P.W. Trinder. A Strategic Profiler for Glasgow Parallel Haskell. In *IFL'98 — Intl. Workshop on the Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 88–102, London, UK, Sep 9–11, September 1998.

[Kit05] NLANR Grid Portal Development Kit, 2005. <URL:`http://dast.nlanr.net/Features/GridPortal/`>.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall International, 1988.

[KTF03] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: a grid-enabled implementation of the message passing interface. *Journal Parallel Distributed Computing*, 63(5):551–563, 2003.

[LB96] B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multithreaded Programming.* Prentice Hall, 1996. ISBN 0-3443698-9.

[LDG+01] X. Leroy, D. Doligez, J. Garrigue, D. Reḿy, and D. Vouillon. The object caml system release 3.02 document and user manual. Technical report, Intitut National de Recherche en Informatique et en Automatique, 2001.

[LH96] H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96 — Intl. Workshop on the Implementation of Functional Languages*, LNCS 1268, pages 184–199, Bonn/Bad-Godesberg, Germany, September 1996. Springer.

[Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[Loi01a] H-W. Loidl. Glasgow Parallel Haskell. WWW page, January 2001. <URL:`http://www.cee.hw.ac.uk/~dsg/gph/`>.

[Loi01b] H-W. Loidl. The Efficiency of Parallel Graph Reduction on a Loosely-coupled Multiprocessor. In *SFP'01 — Scottish Functional Programming Workshop*, Trends in Functional Programming, Stirling, Scotland, August 22–24, 2001, 2001. Intellect.

[Loi02a] H-W. Loidl. Load balancing in a parallel graph reducer. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 63–74, Bristol, UK, 2002. Intellect.

[Loi02b] H-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *CCGrid 2002 — Intl. Symp. on Cluster Computing and the Grid*, pages 311–318, Berlin, Germany, May 2002. IEEE Press.

[Loo99] R. Loogen. Programming Language Constructs. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 3, pages 63–92. Springer-Verlag, 1999.

[LRS+03] H-W. Loidl, F. Rubio Diez, N.R. Scaife, K. Hammon d, U. Klusik, R. Loogen, G.J. Michaelson, R. Horiguchi, S. a nd Pena Mari, S.M. Priebe, A.J. Rebon Portillo, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3):203–251, 2003.

[LTB01] H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, December 2001.

[MHC99] G. Michaelson, K. Hammond, and C. Clack. Foundations. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 2, pages 31–61. Springer-Verlag, 1999.

[Mis94] J. Misra. A Structure for Parallel Recursion. *ACM TOPLAS*, 16(6), 1994.

[MMF+93] C. Mechoso, C.-C. Ma, J. Farrara, J. Spahr, and R. Moore. Parallelization and distribution of a coupled atmosphere-ocean general circulation model. *Monthly Weather Review*, 121(7):2062–2076, 1993.

[MS99] M. Migliardi and V. S. Sunderam. PVM Emulation in the Harness Metacomputing System: A Plug-in Based Approach. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 117–124, London, UK, 1999. Springer-Verlag.

[MSBK01] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, 2001. Special Issue on High Level Models and Languages for Parallel Processing.

[Mur03] T. Murphy VII. Hemlock and the ConCert v2 Framework. Talk at Carnegie Mellon University, 2003.

[Nap05] Napster, 2005. <URL:`http://www.napster.com/`>.

[NBG+96] M. Norman, P. Beckman, Bryan G., J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. Ostriker, J. Shalf, J. Welling, and S. Yang. Galaxies Collide on the I-WAY: An Example of Heterogeneous Wide-Area Collaborative Supercomputing. *International Journal of Supercomputer Applications*, 10(2/3):132–144, 1996.

[ND78] K. Nygaard and O. J. Dahl. The development of the SIMULA languages. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, pages 245–272, New York, NY, USA, 1978. ACM Press.

[NH96] J. Nieplocha and R. J. Harrison. Shared Memory NUMA Programming on I-WAY. In *Proc. of the Fifth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-5)*, 1996.

[Nie99] F. Nielson. Validating Programs in Concurrent ML. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 17, pages 361–377. Springer-Verlag, 1999.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th ACM Annual Symposium on Computer Architecture*, pages 156–167, 1992.

[Par91] A. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*. PhD thesis, University of Tasmania, 1991.

[PBS05] Portable Batch System PBS, 2005. <URL:`http://www.openpbs.org/`>.

[Pey96] S.L. Peyton Jones. Compiling Haskell by Program Transformation: a Report from the Trenches. In *ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Linköping, Sweden, April 22–24, 1996. Springer.

[PHA+99] S.L. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A Non-strict, Purely Functional Language. Electronic document available on-line at http://www.haskell.org/, February 1999.

[PHH+93] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, UK, March 1993.

[PJCSH87] S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *Intl. Conf. on Functional Programming Languages and Computer Architecture*, LNCS 274, pages 98–112, Portland, Oregon, 1987. Springer-Verlag.

[RM01] A. Rajasekar and R. Moore. Data and Metadata Collections for Scientific Applications. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 72–80, London, UK, 2001. Springer-Verlag.

[Roe91] P. Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computer Science, University of Glasgow, 1991.

[Ros98] L. J. Rosenberg. Implementation and Evaluation of Load Distributing Techniques Applied to a Java-Capable Locally Distributed System. Technical report, School of Electrical Engineering and Computer Science, University of Tasmania, 1998.

[Roy01] A. J. Roy. *End-to-end quality of service for high-end applications*. PhD thesis, Argonne National Laboratory, 2001.

[RSL05] Resource Specification Language RSL, 2005. <URL:`http://www.globus.org/gram/rsl_spec1.html/`>.

[SDS05] GridPort Toolkit SDSC, 2005. <URL:`http://www.gridport.npaci.edu/`>.

[Ser99] J. Serot. Explicit Parallelism. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 18, pages 379–396. Springer-Verlag, 1999.

[SGE] Sun Grid Engine SGE. <URL:`http://www.sun.com/software/Gridware/`>.

[SK03] G. Sipos and P. Kacsuk. Executing and Monitoring PVM Programs in Computational Grids with Jini. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 570–576. Springer, 2003. http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2840&spage=570.

[SKH95] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[SKS92] N.G. Shivaratri, P. Krueger, and M. Singhal. Load Distribution for Locally Distributed Systems. *Journal of Computer*, 25(12):33–44, December 1992.

[SMH01] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler. In *IFL'01 — Intl. Workshop on the Implementation of Functional Languages*, volume 2312 of *LNCS*, pages 138–154, Stockholm, Sweden, Sep 24–26, 2001.

[SP95] P.M. Sansom and S.L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In *POPL'95 — Symp. on Principles of Programming Languages*, pages 355–366, San Francisco, CA, Jan 23–25, 1995. ACM Press.

[Str85] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1985.

[SWP90] B. Shirazi, M. Wang, and G. Pathak. Analysis and Evaluation of Heuristic Methods for Static Task Scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.

[THM+96] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.

[TL+00] P.W. Trinder, H-W. Loidl, et al. The Multi-Architecture Performance of the Parallel Functional Language GPH. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, LNCS 1900, pages 739–743, Munich, Germany, 2000. Springer-Verlag.

[TLP02] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, July 2002. Special Issue on Haskell.

[TPL00] P.W. Trinder, R.F. Pointon, and H-W. Loidl. Towards Runtime System Level Fault Tolerance for a Distributed Functional Language. In *SFP'00 — Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, pages 103–113, St Andrews, Scotland, Jul 26–28, 2000. Intellect.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and*

computer architecture, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Tur86] D. Turner. An overview of Miranda. *SIGPLAN Not.*, 21(12):158–166, 1986.

[UDD05] UDDI, 2005. <URL:`http://www.uddi.org/`>.

[Wal] M. Walmsley. *Multi-Threaded Programming in C++*. Springer. ISBN 1-85233-146-1.

[WFK+04] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, and F. Meder, S.and Siebenlist. X.509 Proxy Certificates for Dynamic Delegation. In *3rd Annual PKI R&D Workshop*, 2004.

[WM85] Y-T. Wang and R. J. T Morris. Load Sharing in Distributed Systems . *IEEE Transactions on Software Engineering*, 34(3):204–217, 1985.

[WM95] Y-T. Wang and R. J. T. Morris. Load Sharing in Distributed Systems . In A. Shirazi, A. R. Hurson, and K. M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Transactions on Software Engineering, pages 7–20. ACM, 1995.

[Wol98] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Journal of Cluster Computing*, 1:119–132, January 1998.

[WSD05] Web Services Desciption Langauge WSDL, 2005. Version 1.1 W3C Note 15, March, 2001, <URL:`http://www.w3.org/TR/wsdl/`>.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid Services. In *In International Symposium High Performance Distributed Computing*, pages 48–57, Seattle, WA, June 2003.

[ZFS03] X. Zhang, J. L. Freschl, and J. M. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *HPDC'03:*

Proceeding of the 12th IEEE International Symposium on High perfromance Distributed Computing, pages 270–282, Washington, DCm, USA, 2003. IEEE Computer Society.

[ZHU02] Y. ZHU. A Survey on Grid Scheduling System. Technical report, Department of Computer Science, Hong Kong University of Science and Technology, 2002.

[ZKA04] H. Zhang, K. Keahey, and W. E. Allcock. Providing Data Transfer with QoS as Agreement-Based Service. In *International Conference on Services Computing*, pages 344–353, Shanghai, China, September 2004. IEEE Computer Society.

[ZZWD93] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software - Practise and Experience*, 23(12):1305–1336, 1993.