GRID superscalar: a programming model for the Grid

by

Raül Sirvent Pardell

Advisor:

Rosa Maria Badia Sala

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR PER LA UNIVERSITAT POLITÈCNICA DE CATALUNYA

COMPUTER ARCHITECTURE DEPARTMENT (DAC)
TECHNICAL UNIVERSITY OF CATALONIA (UPC)

Barcelona (Spain)
February 2009

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: GRID superscalar: a programming model for the Grid
Autor de la tesi: Raül Sirvent Pardell

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, …………… de/d'…...............……………. de ....….

El President	El Secretari	
.... (nom i cognoms)	.... (nom i cognoms)	
El vocal	El vocal	El vocal
.... (nom i cognoms)	.... (nom i cognoms)	.... (nom i cognoms)

Abstract

During last years, the Grid has emerged as a new platform for distributed computing. The Grid technology allows to join different resources from different administrative domains and form a virtual supercomputer with all of them. Many research groups have dedicated their efforts to develop a set of basic services to offer a Grid middleware: a layer that enables the use of the Grid. Anyway, using these services is not an easy task for many end users, even more if their expertise is not related to computer science. This has a negative influence in the adoption of the Grid technology by the scientific community. They see it as a powerful technology but very difficult to exploit. In order to ease the way the Grid must be used, there is a need for an extra layer which hides all the complexity of the Grid, and allows users to program or port their applications in an easy way.

There has been many proposals of programming tools for the Grid. In this thesis we give an overview on some of them, and we can see that there exist both Grid-aware and Grid-unaware environments (programmed with or without specifying details of the Grid respectively). Besides, very few existing tools can exploit the implicit parallelism of the application and in the majority of them, the user must define the parallelism explicitly. Another important feature we consider is if they are based in widely used programming languages (as C++ or Java), so the adoption is easier for end users.

In this thesis, our main objective has been to create a programming model for the Grid based on sequential programming and well-known imperative programming languages, able to exploit the implicit parallelism of applications and to speed them up by using the Grid resources concurrently. Moreover, because the Grid has a distributed, heterogeneous and dynamic nature and also because the number of resources that form a Grid can be very big, the probability that an error arises during an application's execution is big. Thus, another of our objectives has been to automatically deal with any type of errors which may arise during the execution of the application (application related or Grid related).

GRID superscalar (GRIDSs), the main contribution of this thesis, is a programming model that achieves these mentioned objectives by providing a very small and simple interface and a runtime that is able to execute in parallel the code provided using the

Grid. Our programming interface allows a user to program a *Grid-unaware* application with already known and popular imperative languages (such as C/C++, Java, Perl or Shell script) and in a sequential fashion, therefore giving an important step to assist end users in the adoption of the Grid technology.

We have applied our knowledge from computer architecture and microprocessor design to the GRIDSs runtime. As it is done in a superscalar processor, the GRIDSs runtime system is able to perform a data dependence analysis between the tasks that form an application, and to apply renaming techniques in order to increase its parallelism. GRIDSs generates automatically from user's main code a graph describing the data dependencies in the application. We present real use cases of the programming model in the fields of computational chemistry and bioinformatics, which demonstrate that our objectives have been achieved.

Finally, we have studied the application of several fault detection and treatment techniques: checkpointing, task retry and task replication. Our proposal is to provide an environment able to deal with all types of failures, transparently for the user whenever possible. The main advantage in implementing these mechanisms at the programming model level is that application-level knowledge can be exploited in order to dynamically create a fault tolerance strategy for each application, and avoiding to introduce overhead in error-free environments.

Acknowledgements

I believe the story of how this thesis was made is quite interesting, and I would like to explain it to you in more detail. It all began in November 2001, when I was finishing my degree in Computer Science and I was looking for a subject for my Graduate Thesis. I met Rosa Maria Badia and Jesús Labarta, who were doing research for CEPBA (European Center of Parallelism of Barcelona) at that time and I liked their proposals. They talked to me about Grid technology, and, although it was also a new topic for them, they faced it with all their research experience and professionalism. I will always remember them and be grateful to them for giving me this first opportunity. It was just at that moment that I started to feel that I liked research.

As soon as I started working as a scholarship holder at CEPBA, I realised that the working environment was very different from that of a normal company. More than a common working place, the CEPBA room was like a dynamic collaborative environment, where everyone was contributing with his best, and everybody together made things advance in an incredible way. If anyone had a doubt (existential or not), their colleagues showed interested in it, and contributed with their opinion or experience, even when their work was not related to it (in 30 square meters there were people working in completely different subjects). This mixture of different areas of expertise enriched your work with multiple points of views, and I believe it was one of the points that made CEPBA such a strong research center.

I have to admit that I didn't plan to do a PhD when I started working at CEPBA. I just wanted to do my Graduate Thesis, get my degree and look for a job in a company. But the working atmosphere, the way everybody worked together, and the subjects we were dealing with, made me enroll in the Computer Architecture Department's PhD program. And, finally, here I am, at the end of the road :). So, what started as a meeting where a student was looking for a Graduate Thesis, ended up with the creation of a group dedicated to Grid computing and clusters inside the Barcelona Supercomputing Center, lead by Rosa Maria Badia. I don't want to seem big-headed, but to have participated in the creation of this group from those early stages makes me feel proud.

Of course, this thesis wouldn't have been possible without the people I have been in touch with, from CEPBA days to the current date. That is why I would like to dedicate some lines to them. If I have forgotten anyone, I apologize, because during these years I have met so many people that it's impossible to name all of them here.

First of all I want to thank my thesis advisor, Rosa Maria Badia. Her experience and advice have been key to successfully develop this thesis. Our professional relationship has been great, intense from time to time, but she has always been very kind to me. Second, to Jesús Labarta, because he was very involved in the first half of the thesis and, because of that, he could have figured as co-advisor undoubtedly.

I will always remember my time at CEPBA, sharing an office with Dani Caldentey, Carles López, Àlex Duran, Francisco Martínez, Montse Farreras, Rogeli Grima, David Carrera, Roman Roset, Carles Tarsà and Juanjo Costa. We were always laughing and there was a great atmosphere in the room. Any visitor might have got the impression that nobody was working in there, but that was far from being true. Ok, from time to time we played a couple of games with XBlast ;), but when there was a deadline everybody worked until the work was finished, without looking at the clock. This excellent working environment and team spirit made it possible for us to finish even the hardest jobs in an incredibly easy way. Some of these colleagues left, others continued at UPC or BSC, but, even when we have chosen different paths, I believe there is a special connection between us that will remain over the years.

In my next time period at CEPBA, when I became a veteran, more people came. I have shared everyday life with many of them, having breakfast, lunch, and cofee breaks together :). People such as Xavi Aguilar, Vicenç Beltran, Jario Balart, David Ródenas, Ivan Rodero and Francesc Guim. With the last two I have had a great relationship, because we were all working on our corresponding PhDs at the same time on the same subject (Grid), and we shared some unique moments (I still remember at Lausanne, when Ivan was trying to talk in French to two girls, and Francesc and I had a good laugh :)).

I don't want to forget those who have been involved with more or less intensity in the GRID superscalar project: Josep Maria Pérez, Vasilis Dialinos, Pieter Bellens, Jorge Ejarque and Marc de Palol. All of them have contributed to the project, making it possible for the project to advance and to reach its objectives.

I want to thank Julita Corbalán for her help in the last part of the thesis. I believe her comments during the preliminary thesis reading with the department helped me to improve the presentation of the work, which is very important. It is not only enough to do a good job, but you also have to present it well.

There is a group of friends that are very special for me. I believe they already know, but from here I want to remind them about it. Ignasi Mulet and Gemma López, who recently have had a baby (Ingrid) :), Sergi Morales and Marta Pla, and Carlos Toledo. Our friendship begun at the university, and it remains as years go by. We always have a lot of fun when we meet, playing Wii or just talking.

Finally, my most special acknowledgement goes to my family. To my parents, Francesc and Concepció, who made me what I am and who have always supported me to keep going. To my sister Laura, who has been always a source of inspiration to all her brothers and sisters. Lately, she has been a bit distant from the family, but it is nothing that we cannot solve with a couple of weekends together, as she has always been like a second mother to all of us. To my sister Núria, who read stories to me when I was a child and, maybe she is not fully aware of it, but she is a person whose incredible strength and perseverance ensures that she achieves what she wants. To my niece Paula, whose energy spreads to everybody around her. To my twin brother David, to whom I believe nothing more needs to be added than saying that we are twin brothers ;). To his wife Laura, because she already suspected that marrying a twin brother was like marrying two persons :D.

To everybody, thank you.

Raül

This work has been partially funded by the Ministry of Science and Technology of Spain (contracts CICYT TIC2001-0995-CO2-01, CICYT TIN2004-07739-C02-01, CICYT TIN2007-60625) by the CoreGRID European Network of Excellence (contract IST-2002-004265) and by the BEinGRID EU project (contract IST-2005-034702).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Grid computing is a buzzword coined in the 1990s by Ian Foster to describe a collection of geographically distributed resources connected through a wide area network [34]. The resources are not only computers, but also storage, sensors, PDAs, etc. In such an organization, the Grid appears to an end user as one large virtual system. The objective is not only to provide computing power but also to allow access to data or enable collaboration. In the last years, the Grid has entered the scene as a new computing alternative and it has become a very important research and development area in this decade. As we have seen, the Grid is characterized for having its resources geographically distributed in a large distance area. Thus, the set of problems tackled by the Grid scientific community is very large, and most of them are not directly related with computation intensive applications. However, the possibility of using all the computing elements connected by the Grid as a unique and very large supercomputer is very attractive.

A lot of effort has been devoted by the Grid middleware community to build basic middleware systems, such as the Globus Toolkit [33], Condor [37] or Unicore [31] just to cite the more well known. These initiatives have come up with Grid services that allow to perform actions in a computational Grid such as job submission, file transfer, resource discovering and so on. However, it is still very difficult for non expert users to deal with a Grid environment by directly using these services. It is not reasonable that scientists have to discover and select the Grid resources, prepare scripts to submit a collection of jobs to remote hosts, or take care of transferring the files. This may not be difficult for a single job, but it becomes extremely complex for a big set of jobs that must be run in the Grid.

The basic Grid architecture layers are shown in Figure 1.1. An application must use the Grid middleware services in order to exploit the remote resources available in the

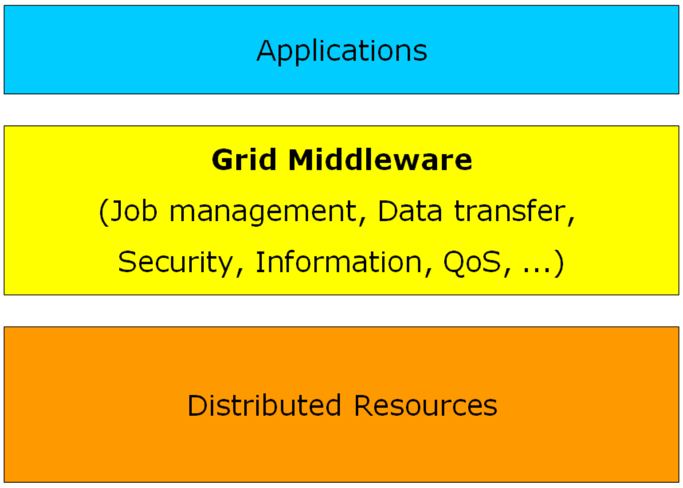

Applications
Grid Middleware
(Job management, Data transfer,
Security, Information, QoS, ...)
Distributed Resources

Figure 1.1 Grid architecture layers

Grid. However, these services are complex to be used directly, as they are a pack of very diverse services that allow to control many different things in the Grid: job management, data transfer, security, information services, and so on. In order to hide the complexity of using middleware services directly, many research groups have worked in building programming environments, tools and systems in order to ease the way these services can be used. Adding an extra layer between the application and the Grid middleware is also useful to avoid an application to be bound to a particular middleware: users can develop their applications with a programming environment and the programming environment is the one that needs to care about being able to work with different Grid middlewares.

The parallel execution of complex applications has proved to be an excellent way of reducing the time to get results. By running this applications in large HPC facilities allows the scientific community to afford scientific challenges that were impossible to imagine some years ago. The most well known paradigms for parallel programming are message passing and shared memory. The MPI standard [66] has been extensively used by the scientific community in the recent years. Similarly, OpenMP [71] can be used for shared memory architectures. However, the parallelization is not for free: the scientific community has invested lots of efforts in porting or writing the applications in these programming models. This is specially true with MPI, since its programming interface is not as friendly as a non-expert programmer would desire. OpenMP is a much more programmable paradigm, but currently is is bound to shared memory architectures. However, distributed shared memory (DSM) versions are currently being developed by different groups [25].

Another important concern of Grid computing is if a killer application[1] will appear or not. This concern comes partially because of the difficulty of writing applications for a computational Grid, the same concern as with MPI and OpenMP. Although skilled programmers may be willing and able to write applications with complex programming models, scientists usually expect to find easy programming methodologies that allow them to develop their applications with both flexibility and ease of use. Similarly to parallel programming, programming models for the Grid can become crucial to ensure a successful story.

We can find in the literature many systems that try to help users to execute an application in the Grid. The main characteristics that we find interesting in these tools are the capacity of being non conscious of the Grid and to exploit the parallelism of an application implicitly. A programming model is considered to be *Grid-unaware* when a user does not have to specify any information about how a particular application is going to be run in the Grid. The system is in charge of determining how the Grid resources are going to be exploited. Regarding parallelism, we talk about *explicit parallelism* when users program their application in a parallel fashion, thus specifying which parts of the application can be executed in parallel, and when this parallelism must be stopped. This is opposed to the *implicit parallelism* concept, that is used when programmers do not need to detail the parallelism strategy for their application. In order to achieve simplicity in programming the Grid, the most interesting concepts are to provide a Grid-unaware programming model which is able to implicitly exploit the parallelism of the application. This way an application can be executed in a parallel way in the Grid, while the user does not necessary need to program it with a parallel programming paradigm.

Another key aspect which eases the adoption of a programming model is the language used to program with it. Some systems found in the literature create a new procedural language to develop applications for the Grid. However, this makes mandatory for users to learn this new language, which is an extra barrier for the adoption of the technology. In contrast, using popular programming languages as the input of the programming model, such as C++, Java, and so on, can contribute to make easier the step that users must do to work with the Grid.

There also exist other interesting features to add to a programming model for the Grid. The Grid, as a distributed, dynamic and heterogeneous environment prone to failures: an execution in a Grid can use hundreds or thousands of machines distributed worldwide, in different administrative domains, etc. Thinking that in such a distributed environment no errors will arise is very difficult. The ability of facing all these errors is what is called

[1]A determining application that causes the massive adoption of a technology

fault tolerance. Fault tolerance is applied with different objectives and at different levels in existing solutions. Failures can be transient, intermittent or permanent, and they may be related to the application's code (memory leaks, ...) or the underlying Grid services. There are some systems which deal with application related failures and some others deal with errors in the Grid middleware services offered, or in the remote machines. Some others try to integrate both types of failures, but require that the user specifies what to do when a failure arises, and this is a difficult task for the end user.

1.2 Thesis objectives

Our **main objective** is to create a programming model for the Grid. As many other already created programming models, it will avoid that the user must deal directly with Grid middleware services. We want to provide a very simple interface and focus it to **sequential programming**, to make it very easy to use. Its runtime must be able to parallelize the sequential application programmed, by detecting and exploiting the **implicit parallelism** of the application. This means that the sequential application will be transformed in a workflow based application, where the nodes are parts of the application that must be executed in the Grid (that we will call tasks), and the edges are data dependencies between those nodes. The parallel execution of the application will allow to **speed it up** in the Grid, always depending on the resources available and the possibilities inside the application to be executed in parallel.

This environment must keep the Grid as invisible as possible to the end user. The application must be programmed in a **Grid-unaware** fashion, thus, not specifying any details about where a task must be run, or where data must be transferred in the Grid. The reference programming languages for us will be well-known **imperative languages**, such as C/C++, Java, Perl and Shell Script. Basing our programming model in them allows to ensure an easy adoption for end users, as they can use their preferred programming language to work with our system.

In our system we consider that the type of applications which are suitable to be executed in the Grid are those that are composed of coarse grained tasks. These tasks can be of the size of a simulation, program, solver, ... These kinds of applications are very common in bioinformatics, computational chemistry and other scientific fields. Examples of these applications are Monte Carlo simulations, optimization algorithms, parametric studies or complex workflows where each element of the workflow is a computational intensive task. The data that these applications use commonly as inputs or outputs are files, because sometimes the data is so big that it cannot be placed all in memory at the same time. An example of this is the technique named *out-of-core* that allows to work with big matrixes that do not fit into main memory. They are divided into smaller sub-matrixes called blocks that fit into memory, and are used to operate. We will consider files as the main exchange of data between tasks, in order to read inputs to perform calculations, or to store results.

The main requirement for our programming model is the Grid middleware, which provides the basic services we need in order to run parts of the application in a Grid environment. Our system needs job management services (submission and control of jobs), file management services (transfer and deletion of files) and security services (authentication and authorisation). We may use a particular middleware in order to develop our system, but we do not want to be bound to a single one. Therefore, we will design our system keeping in mind that we want to be able to use other Grid middlewares that may appear in the future.

We want also to integrate the **fault detection and recovery** at the programming model level. The main advantage in treating failures in this level is a better knowledge of the application, allowing to treat failures in a different manner when we work with different applications. The mechanisms we want to investigate are **checkpointing, task retry and task replication**.

Checkpointing is defined as the ability to save the work performed by an application before a failure occurs. This failure can be detected at the application level or the Grid level (middleware or machines). In case of failure, the application can be restarted in the moment it was checkpointed, thus, avoiding to repeat any computations that were previously done.

In order to allow an application to keep running despite failures, the most commonly used mechanism is the retry of tasks. Whenever a Grid level failure is detected, the task can be repeated in the same or a new resource. Several research groups have included different strategies to retry tasks, however some are not able to work with workflow applications, detect performance degradations, or decide dynamically where to retry tasks.

If we want to overcome the time needed to resubmit a task from the beginning, task replication is a good solution. The key point in task replication is that free resources can execute copies of tasks in order to ensure their execution: if one of the replicas fails, the rest keep running and thus we can get the results of a single task from any of them. Another interesting feature about replication is the possibility to use it in heterogeneous Grids with the objective of using faster resources when they become available: a task may have been initially assigned to a resource, and after that, a faster resource becomes available. The common strategy to use those faster resources is task migration, which consists in checkpointing a task in the slow resource, and restart the task in the fast resource. This is very effective to achieve a better execution time for the application, however it requires an intra-task checkpoint mechanism implemented by the user.

All these mechanisms must be designed to be as transparent as possible to end users and to cause the minimum overhead to the system when no failures are detected. This is to be consistent with our objectives of providing a very simple interface and to achieve speed up when executing applications.

1.3 Thesis contribution

The main contribution of this thesis is GRID superscalar (GRIDSs), a programming model that eases the development of Grid applications to the point that writing such an application can be as simple as programming a sequential program to be run on a single processor and the hardware resources remain totally transparent to the programmer.

GRID superscalar is based on how superscalar processors execute assembler codes [47]. Even though the assembler codes are sequential, the implicit parallelism of the code is exploited in order to take advantage of the functional units of the processor. The processor explores the concurrency of the instructions and assigns them to the functional units. Indeed, the execution order defined in the sequential assembler code may not be followed. The processor will establish the mechanisms to guarantee that the results of the program remain identical. If the performance achieved by the application is better than the one that would have been initially obtained, the programmers will be grateful. All these ideas are exportable to the Grid application level. What changes is the level of granularity: in the processors we have instructions lasting in the order of nanoseconds and in computational Grids, functions or programs that may last from some seconds to hours. Also, what it changes is the objects: in assembler the objects are registers or memory positions, while in GRID superscalar we will deal with files, similar to scripting languages.

The first contribution of this thesis is the **GRID superscalar programming interface**. We pretend that our system acts as an "intermediate language", as the assembler language does between programming languages and machine language. On top of it, users can program with different languages, such as C, C++ or Java. From those programs, an intermediate representation of the application is generated (in GRIDSs a direct acyclic graph with the data dependencies between the tasks that form the application). And from that intermediate representation, the application is executed. An overview of this behavior is presented in Figure 1.2. Following the simile, the tasks defined to be executed in the Grid could be considered as the Instruction Set of our "assembler language".

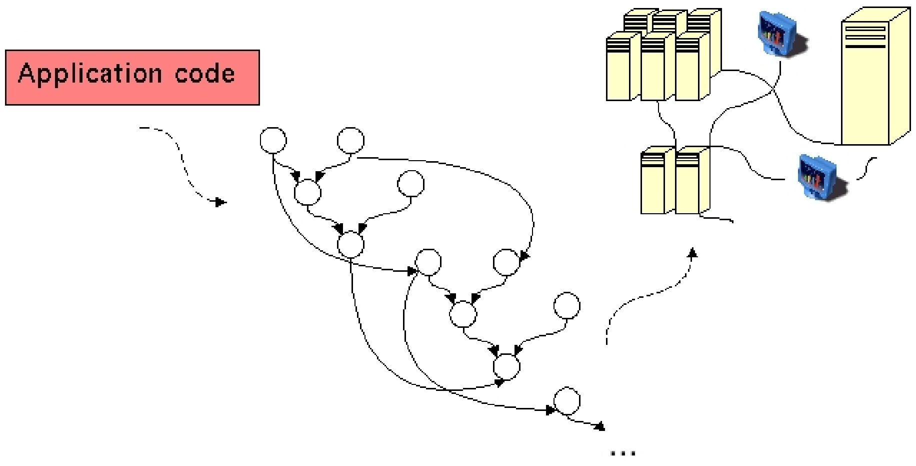


Figure 1.2 Overview of GRID superscalar behavior

An application written with GRIDSs follows the master-worker parallel programming paradigm. The main part of the application is the master, that will automatically generate tasks for the runtime when the programmed algorithm is executed. The functions that must be executed in the Grid will be executed by creating a worker in a remote machine that will call to the corresponding function with the correct parameters. So, we can consider GRIDSs applications master-worker applications, even though GRIDSs API is not a master-worker API, because programmers are not in charge of splitting and distributing the work from the master to the workers, thus the master-worker structure is hidden to the application programmer.

Several steps are mandatory to program an application with GRIDSs. The application developer provides a sequential program, composed of a main program with calls to application tasks, together with an Interface Description Language (IDL) file. As it has been stated previously, our programming model takes sequential programming as a reference. This allows to build a very small an simple API to be used in the applications. The API is composed of 6 calls which can be used in the master, and 2 in the worker. Despite the simplicity of the API, several advanced features can be specified, such as the speculative execution of tasks and the specification of constraints and cost for tasks. In the IDL file the user must specify the functions that must be run in the Grid, together with their parameters. This IDL file is not only used to know the type and direction of the parameters in the functions, but also to build a master-worker application.

The second contribution of this thesis is the **GRID superscalar runtime**. The behavior of the application when run with GRID superscalar is the following: for each task candidate to be run in the Grid, the GRID superscalar runtime inserts a node in a task graph. Then, the GRID superscalar runtime system seeks for data dependencies between the different tasks of the graph. These data dependencies are defined by the input/output of the tasks which are files. If a task does not have any dependence with previous tasks which have not been finished or which are still running (i.e., the task is not waiting for any data that has not been already generated), it can be submitted for execution to the Grid. An advanced technique known as *renaming* [47] is applied in the task graph in order to eliminate some data dependencies between tasks and therefore increase the possibilities of executing tasks in parallel. This technique is a unique feature only available in GRID superscalar, and it allows to solve concurrency problems when two tasks want to write in the same files, or when a task must overwrite some files that other tasks must read.

Although our objective is not to provide a complex resource broker, this feature is needed in the runtime. Therefore, we provide a simple broker which will select between the set of available hosts, which is the best suited for a task. This selection favours reducing the total execution time, which is computed not only as the estimated execution time of the task in the host but also the time that will be spent to transfer all files required by the task to the host. This allocation policy exploits the file locality, reducing the total number of file transfers. Those tasks that do not have any data dependence between them can be run on parallel on the Grid. This process is automatically controlled by the GRID superscalar runtime, without any additional effort for the user, as well as all Grid related actions (file transfer, job submission, end of task detection, results collection) which are totally transparent to the user. The GRID superscalar is notified when a task finishes. Next, the data structures are updated and any task than now have its data dependencies resolved, can be submitted for execution.

When the application has finished, all working directories in the remote hosts are cleaned, application output files are collected and copied to their final locations in the localhost and all is left as if the original application has been run. Summarizing, from the user point of view, an execution of a GRID superscalar application looks as an application that has run locally, with the exception that the execution has been hopefully much faster by using the Grid resources.

The publications related to this first two contributions are:

[11] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima, "Programming Grid Applications with GRID Superscalar", Journal of Grid Computing, Volume 1, Issue 2, 2003.

[12] Rosa M. Badia, Raul Sirvent, Jesus Labarta, Josep M. Perez, "Programming the GRID: An Imperative Language-based Approach", Engineering The Grid: Status and Perspective, Section 4, Chapter 12, American Scientific Publishers, January 2006.

[90] Raül Sirvent, Josep M. Pérez, Rosa M. Badia, Jesús Labarta, "Automatic Grid workflow based on imperative programming languages", Concurrency and Computation: Practice and Experience, John Wiley & Sons, vol. 18, no. 10, pp. 1169-1186, 2006.

[86] Raül Sirvent, Rosa M. Badia, Pieter Bellens, Vasilis Dialinos, Jesús Labarta, Josep M. Pérez, "Demostración de uso de GRID superscalar", Boletín RedIRIS, no. 80, Abril 2007.

[91] Raül Sirvent, Josep M. Pérez, Rosa M. Badia, Jesús Labarta, "Programación en el Grid: una aproximación basada en lenguajes imperativos", Computación Grid: del Middleware a las Aplicaciones, pp. 33-52, Junio 2007.

Finally, the third contribution of the thesis is the **fault tolerance mechanisms** studied in GRID superscalar. We have implemented a tailored inter-task checkpoint mechanism which is able to save the state of a running application transparently to the user and without causing any overhead to the execution time of the application. This is done by recovering the sequential consistency of the application.

Task retries are also used in order to overcome failures at run time without the need to stop the application. Problematic machines can be automatically discarded at run time. Soft and hard timeouts are defined for the tasks to detect errors not notified by the Grid middleware. Results for each task are transferred asynchronously to the master machine in order to keep the checkpoint mechanism consistent. Grid operations are retried in the runtime in case of failure and also other causes of errors in the master are checked.

The last mechanism is the task replication, which pursues to hide the overhead in the execution time which may be caused by a task retry. This is done by replicating the task in advance, thus, in case of a failure in a machine, another one is already running the same task and can obtain its results. Besides, replication is also useful to execute tasks in faster resources when they become available (instead of the ones initially used) without the need of implementing an intra-task checkpoint, which is commonly left as a responsibility for the user in most systems. We only submit replicas to the system when free machines in the Grid are available. Thus, sequential parts of an application will be replicated to ensure they do not delay the whole execution, and highly parallel parts of the application will not be replicated to avoid overloading the resources excessively.

The publications related to this contribution are:

[29] Vasilis Dialinos, Rosa M. Badia, Raül Sirvent, Josep M. Pérez and Jesús Labarta, "Implementing Phylogenetic Inference with GRID superscalar", Cluster Computing and Grid 2005 (CCGRID 2005), Cardiff, UK, 2005.

[88] Raül Sirvent, Rosa M. Badia and Jesús Labarta, "Fault tolerance and execution speed up with task replication for scientific workflows", High Performance Computing and Communications (HPCC) (Submitted), 2009.

Moreover, our contributions have also derived in the following work:

[89] Raül Sirvent, Andre Merzky, Rosa M. Badia, Thilo Kielmann, "GRID superscalar and SAGA: forming a high-level and platform-independent Grid programming environment", CoreGRID Integration Workshop. Integrated Research in Grid Computing, Pisa (Italy), 2005.

[5] Rosa M. Badia, Pieter Bellens, Vasilis Dialinos, Jorge Ejarque, Jesús Labarta, Josep M. Pérez and Raül Sirvent, "The GRID superscalar project: Current status", XVII Jornadas de Paralelismo. Albacete (Spain), September 2006.

[8] Rosa M. Badia, Pieter Bellens, Marc de Palol, Jorge Ejarque, Jesús Labarta, Josep M. Pérez, Raül Sirvent and Enric Tejedor, "GRID superscalar: from the Grid to the Cell processor", Spanish Conference on e-Science Grid Computing. Madrid (Spain), March 2007.

[9] R. M. Badia, P. Bellens, J. Ejarque, J. Labarta, M. de Palol, J. M. Pérez, R. Sirvent and E. Tejedor, "SSH GRID superscalar: a Tailored Version for Clusters", 1st Iberian Grid Infrastructure Conference (IBERGRID). Santiago de Compostela (Spain), May 2007.

[80] S. Reyes, A. Niño, C. Muñoz-Caro, R. Sirvent and R. M. Badia, "Monitoring Large Sets of Jobs in Internet-Based Grids of Computers", 1st Iberian Grid Infrastructure Conference (IBERGRID). Santiago de Compostela (Spain), May 2007.

[62] Róbert Lovas, Raül Sirvent, Gergely Sipos, Josep M. Pérez, Rosa M. Badia, Péter Kacsuk, "GRID superscalar enabled P-GRADE portal", Integrated Research in GRID Computing, Springer, November 2007.

[87] Raül Sirvent, Rosa M. Badia, Natalia Currle-Linde, Michael Resch, "GRID superscalar and GriCoL: integrating different programming approaches", Achievements in European Research on Grid Systems, Pages 139-150, Springer, January 2008.

[58] Elzbieta Krepska, Thilo Kielmann, Raül Sirvent, Rosa M. Badia, "A service for reliable execution of Grid applications", Achievements in European Research on Grid Systems, Pages 179-192, Springer, January 2008.

[7] Rosa M. Badia, Raül Sirvent, Marian Bubak, Wlodzimierz Funika, Cezary Klus, Piotr Machner, Marcin Smetek, "Performance monitoring of GRID superscalar with OCM-G/G-PM: integration issues", Achievements in European Research on Grid Systems, Pages 193-205, Springer, January 2008.

[10] R.M. Badia, D. Du, E. Huedo, A. Kokossis, I. M. Llorente, R. S. Montero, M. de Palol, R. Sirvent, and C. Vázquez, "Integration of GRID superscalar and GridWay Metascheduler with the DRMAA OGF Standard", Euro-Par, 2008.

1.4 Thesis organization

The rest of this thesis document is organized as described here. Chapter 2 shows a big picture of the research work related to the areas that GRID superscalar covers: programming models for the Grid, tools for Grid workflows and tools or environments providing fault tolerance features. Chapter 3 explains in detail the design and features of the GRID superscalar programming interface, as well as a programming comparison between GRIDSs and some other tools. Chapter 4 presents the scientific contributions and developments of the GRID superscalar runtime. as well as a set of experiments done to evaluate the runtime and the scalability of applications. The research done in GRIDSs related to fault tolerance is described in Chapter 5. This includes the checkpoint, task retry and task replication mechanisms. Finally, Chapter 6 explains our conclusions in this thesis, and proposes future work.

Chapter 2

Related work

Different scientific communities (high-energy physics, gravitational-wave physics, geophysics, astronomy, bioinformatics and others) deal with applications with large data sets whose input consists of non monolithic codes composed of standalone application components which can be combined in different ways. Examples of this kind of applications can be found in the field of astronomy where thousands of tasks need to be executed during the identification of galaxy clusters [27]. These kinds of applications can be described as *workflows*. We will see that many systems which try to aid users to program applications in the Grid have dedicated their efforts in giving support to describe and execute workflows, in order to be able to deal with these applications.

Many tools for the development of workflow based applications for Grid environments have been presented in the literature. BPEL4WS [102] is a language to create workflows proposed by the Web Services Choreography Working Group [110] where medium-size graphs can be defined. Chimera [35] is able to generate an abstract workflow from the workflow description given by the user with the VDL language. Pegasus [27] then, can map that abstract workflow to a concrete workflow (with execution details, i.e. for DAGMan [30]). Triana [99] is a visual programming environment where applications can be described as connected components. Its modular design allows to use different languages to specify the workflow, as well as different Grid middlewares to execute it. The Fraunhofer Resource Grid [48] includes a workflow system where the applications are specified with Petri Nets. Taverna [98] stores internally workflows in textual language, but they are created with a graphical interface. GridAnt [2] provides a framework for prototyping Grid applications based on XML specifications. The task dependencies are also specified manually in the XML. Defined recently, Swift [114] combines the SwiftScript language to generate workflows with mapping descriptors, that describe how logical data maps to physical data.

We can see that in all of these systems the user has to specify the task dependence graph in a graphical or a non-imperative language. While in the graphical case it can be very easy for end users to draw the graph that describes the dependencies between tasks, this will not be efficient if the graph describing the dependencies has a big number of nodes and edges that the user has to draw. Other approaches define a new language to allow generating workflows in an algorithm-like way, but users need to learn that language in order to be able to create their application, and that can be a difficult task for them.

Some other approaches can be found in the bibliography that try to tackle the gap between application programmers and Grid services. The Simple API for Grid Applications Working Group [85] has defined the SAGA API [43], a Grid-aware API that abstracts and simplifies the paradigms provided by the various Grid middleware incarnations. SAGA is an extra layer that hides the details of existing or new Grid middlewares, however, its Grid-aware approach leaves the user in charge of many details related to the Grid services. The Grid Remote Procedure Call Working Group [45] has specified a GridRPC API [83]. Examples of implementations for that API are Ninf-G [97] and NetSolve/GridSolve [4]. GridRPC helps users to port their applications to the Grid, but the parallelism of the application must be explicitly described. Satin [105], based in Java, allows to express divide-and-conquer parallelism. The user is in charge of opening and closing parallel regions in the code. Javelin [67] presents a branch-and-bound computational model where the user partitions the total amount of work. AppLeS [17] is a framework for adaptively scheduling applications on the Grid. It also provides a template for master-worker applications. GAF4J [106] is based in Java and allows thread programming, where threads are run in the Grid. Higher-Order Components (HOCs) [1] express recurring patterns of parallelism that are provided to the user as program building blocks, pre-packaged with distributed implementations. ParoC++ [68] is a C++ extension that allows to execute objects in parallel in the Grid, but requires that users define the synchronization points and the mutual exclusion regions in the code. The Grid Programming Environment (GPE) [44] from Intel is completely based in Java and includes not only the GridBean SDK to easily program Grid applications but also a workflow editor where BPEL workflows can be constructed. ProActive [13] is a powerful Grid middleware for Java applications that achieves implicit parallelism inside Active Objects, which can be deployed in the Grid to run concurrently. There also exists Grid-enabled MPI approaches, as the PACX-MPI [72] and MPICH-G2 [56]. They are very useful to execute applications programmed using the MPI interface in the Grid. However, users are in charge of splitting and distributing the work when they program the application.

All of these presented systems require skills from programmers to implement the parallel execution of the application. Moreover, no approach is found in the bibliography that allows to directly parallelize and execute in the Grid sequential applications.

We can see the importance of the thesis main topic in the number of groups working in finding an easy way to program the Grid. The number of tools available for that purpose is very big, thus, in next sections we have highlighted the most important, splitting them in two main categories: tools that their main goal is the definition of workflows (to be executed in the Grid) and programming models to ease the Grid programming.

After that, a section discusses why GRIDSs is a unique framework compared to the rest. It is a programming model based on already existing programming languages (C, C++, Java, ...), which supports several of those well-known languages, which generates automatically a workflow describing the data dependencies between tasks contained in a program, and which executes that workflow concurrently in the Grid using advanced techniques to remove data dependencies between tasks. As we will see in this chapter, there is no tool that contains all these features.

The last section in this chapter explains the existing work in fault tolerance applied to other Grid systems. We split it in two main categories: fault detection and recovery applied at the Grid level (which can only detect Grid level failures), and applied at the application level (which can handle Grid and application errors). Finally, we compare them with the mechanisms provided in the GRID superscalar runtime.

2.1 Workflow tools

2.1.1 Condor DAGMan

A directed acyclic graph (DAG) can be used to represent a set of computations where the input, output, or execution of one or more computations is dependent on one or more other computations. The computations are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor [37] finds computers for the execution of applications, but it does not schedule applications based on dependencies. The Directed Acyclic Graph Manager [30] (DAGMan) is a meta-scheduler for the execution of programs (computations). DAGMan submits the programs to Condor in an order represented by a DAG and processes the results. A DAG input file describes the DAG, and further submit description files are used by DAGMan when submitting programs to run under Condor. As DAGMan submits programs, it monitors log files to enforce the ordering required within the DAG. DAGMan is also responsible for scheduling, recovery, and reporting on the set of programs submitted to Condor. The input file used by DAGMan is called a DAG input file. It may specify:

- A description of the dependencies within the DAG.
- Any processing required to take place before submission of a node's Condor job, or after a node's Condor job has completed execution.
- The number of times to retry a node's execution, if a node within the DAG fails.
- A node's exit value that causes the entire DAG to abort.

Pre and Post scripts are optional for each job, and these scripts are executed on the machine from which the DAG is submitted; this is not necessarily the same machine upon which the node's Condor job is run. Further, a single cluster of Condor jobs may be spread across several machines. A PRE script is commonly used to place files in a staging area for the cluster of jobs to use and a POST script is commonly used to clean up or remove files once the cluster of jobs is finished running (see Figure 2.1). An example uses PRE and POST scripts to stage files that are stored on tape. The PRE script reads compressed input files from the tape drive, and it uncompresses them, placing the input files in the current directory. The cluster of Condor jobs reads these input files. and produces output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged input files and the output files. DAGMan takes note of the exit value of the scripts as well as the job. A script with an exit value different to 0 fails.

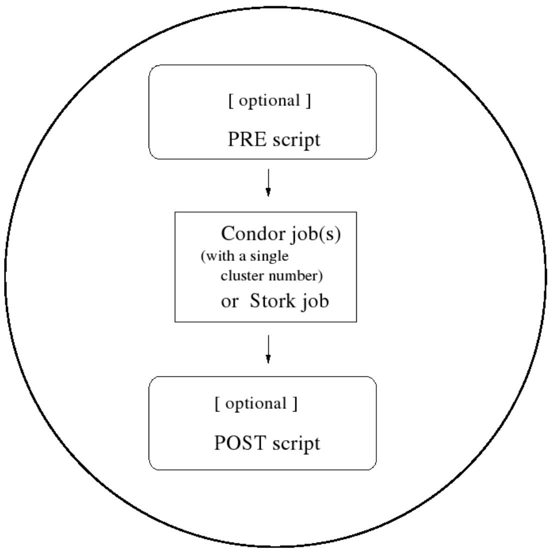


Figure 2.1 Job structure in DAGman

Each node in a DAG may use a unique submit description file. One key limitation is that each Condor submit description file must submit jobs described by a single cluster number. At the present time DAGMan cannot deal with a submit file producing multiple job clusters.

DAGMan can help with the resubmission of uncompleted portions of a DAG, when one or more nodes results in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG. The Rescue DAG is a DAG input file, functionally the same as the original DAG file, but it additionally contains an indication of successfully completed nodes. If the DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be re-executed. The Rescue DAG is automatically generated by DAGMan when a node within the DAG fails. Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

The granularity defining success or failure in the Rescue DAG input file is given for nodes. The Condor job within a node may result in the submission of multiple Condor jobs under a single cluster. If one of the multiple jobs fails, the node fails. Therefore, a resubmission of the Rescue DAG will again result in the submission of the entire cluster of jobs.

The organization and dependencies of the jobs within a DAG are the keys to its utility. There are cases when a DAG is easier to visualize and construct hierarchically, as when a node within a DAG is also a DAG. Condor DAGMan is able to handle this situation. Since more than one DAG is being discussed, terminology is introduced to clarify which DAG is which. To make DAGs within DAGs, the important thing is getting the correct name of the submit description file for the inner DAG within the outer DAG's input file.

Multiple, independent DAGs may be submitted at the same time. Internally, all of the independent DAGs are combined into a single, larger DAG, with no dependencies between the original independent DAGs. As a result, any generated rescue DAG file represents all of the input DAGs as a single DAG. The success or failure of the independent DAGs is well defined. When multiple, independent DAGs are submitted with a single command, the success of the composite DAG is defined as the logical AND of the success of each independent DAG. This implies that failure is defined as the logical OR of the failure of any of the independent DAGs.

2.1.2 Pegasus

In general, a workflow can be described in an abstract form, in which the workflow activities are independent of the Grid resources used to execute the activities. In Pegasus, this workflow is denoted as an abstract workflow (AW). Abstracting away the resource descriptions allows the workflows to be portable. One can describe the workflow in terms of computations that need to take place without identifying particular resources that can perform this computation. Clearly, in a VO (Virtual Organization) environment, this level of abstraction allows for easy sharing of workflow descriptions between VO participants.

Pegasus is designed to map abstract workflows onto the Grid environment. The abstract workflows can be constructed by using Chimera's Virtual Data Language (VDL) [35] or can be written directly by the user. The inputs to the Chimera system are partial workflow descriptions that describe the logical input files, the logical transformations (application components) and their parameters, as well as the logical output files produced by these transformations. Pegasus produces a concrete (executable) workflow that can be given to Condor's DAGMan [30] for execution. Pegasus and DAGMan are able to map and execute workflows on a variety of platforms: condor pools, clusters managed by LSF or PBS, and individual hosts. Figure 2.2 describes the process of workflow generation, mapping and execution.

In the definition of Virtual Data, data refers not only to raw published data, but also data that has been processed in some fashion. Since data processing can be very expensive, sharing these data products can save the expense of performing redundant computations. The concept of Virtual Data was first introduced within the GriPhyN project (www.griphyn.org). An important aspect of Virtual Data are the applications that produce the desired data products. In order to discover and evaluate the validity of the Virtual Data products, the applications are also published within the VO. Pegasus is released as part of the GriPhyN Virtual Data Toolkit and has been used in a variety

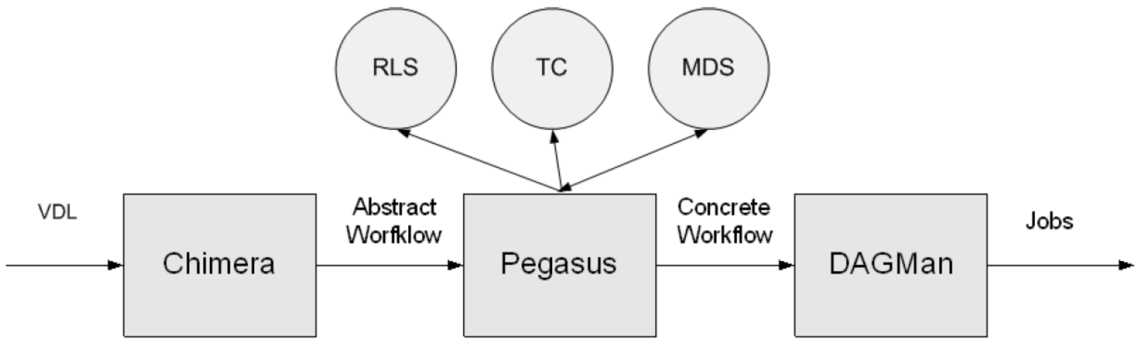


Figure 2.2 Workflow generation, mapping and execution

of applications ranging from astronomy, biology, gravitational-wave science, and high-energy physics.

The GriPhyN Virtual Data System (VDS) that consists of Chimera, Pegasus and DAGMan has been used to successfully execute both large workflows with an order of 100,000 jobs with relatively short run times and workflows with small number of long-running jobs. In the process of workflow generation, mapping and execution, the user specifies the VDL for the desired data products, Chimera builds the corresponding abstract workflow representation. Pegasus maps this AW to its concrete form and DAGMan executes the jobs specified in the Concrete Workflow (CW).

The abstract workflows describe the computation in terms of logical files and logical transformations and indicate the dependencies between the workflow components. Mapping the abstract workflow description to an executable form involves finding the resources that are available and can perform the computations, the data that is used in the workflow, and the necessary software.

Pegasus uses the logical filenames referenced in the workflow to query the Globus Replica Location Service (RLS) [23] to locate the replicas of the required data. It is assumed that data may be replicated in the environment and that the users publish their data products into RLS. Given the set of logical filenames, RLS returns a corresponding set of physical file locations. After Pegasus produces new data products, it registers them into the RLS as well (unless the user does not want that to happen). Intermediate data products can be registered as well.

In order to be able to find the location of the logical transformations defined in the abstract workflow, Pegasus queries the Transformation Catalog (TC) using the logical transformation names. The catalog returns the physical locations of the transformations (on possibly several systems) and the environment variables necessary for the proper execution of the software.

Pegasus queries the Globus Monitoring and Discovery Service (MDS) [26] to find the available resources, their characteristics such as the load, the scheduler queue length, and available disk space. The information from the TC is combined with the MDS information to make scheduling decisions. When making resource assignment, Pegasus prefers to schedule the computation where the data already exist, otherwise it makes a random choice or uses a simple scheduling technique.

Additionally, Pegasus uses MDS to find information about the location of the gridftp servers that can perform data movement, job managers that can schedule jobs on the remote sites, storage locations, where data can be pre-staged, shared execution directories, the RLS where new data can be registered into, site-wide environment variables, etc.

This information is necessary to produce the submit files that describe the necessary data movement, computation and catalog updates.

The information about the available data can be used to optimize the concrete workflow from the point of view of Virtual Data. If data products described within the AW already exist, Pegasus can reuse them and thus reduce the complexity of the CW. In general, the reduction component of Pegasus assumes that it is more costly to execute a component (a job) than to access the results of the component if that data is available. For example, some other user may have already materialized (available on some storage system) part of the entire required dataset. If this information is published into the RLS, Pegasus can utilize this knowledge and obtain the data, thus avoiding possibly costly computation. As a result, some components that appear in the abstract workflow do not appear in the concrete workflow.

One of the key aspects in Pegasus that other systems focus on resource brokerage and scheduling strategies, but Pegasus uses the concept of virtual data and provenance to generate and reduce the workflow based on data products which have already been computed earlier. It prunes the workflow based on the assumption that it is always more costly the compute the data product than to fetch it from an existing location. Pegasus also automates the job of replica selection so that the user does not have to specify the location of the input data files. Pegasus can also map and schedule only portions of the workflow at a time, using just in-time planning techniques.

2.1.3 Triana

Triana [99] is an open source, distributed, platform independent Problem Solving Environment (PSE) written in the Java programming language. A PSE is a complete, integrated computing environment for composing, compiling, and running applications in a specific application domain. The Triana PSE is a graphical interactive environment that allows users to compose applications and specify their distributed behavior. A user creates a workflow by dragging the desired units from a toolbox onto the workspace and interconnects them by dragging a cable between them.

Triana includes the mechanisms which are the basic requirements of a framework for building complex distributed Web service based systems. By facilitating the transparent construction of Web services workflows, users can focus on the design of the workflow at a conceptual level, easily create new complex composite services (workflows made of services) which offer more functionality than available atomic services, easily expose composite services enabling users to share and replicate experiments, and easily carry out ’what-if’ analysis by altering existing workflows.

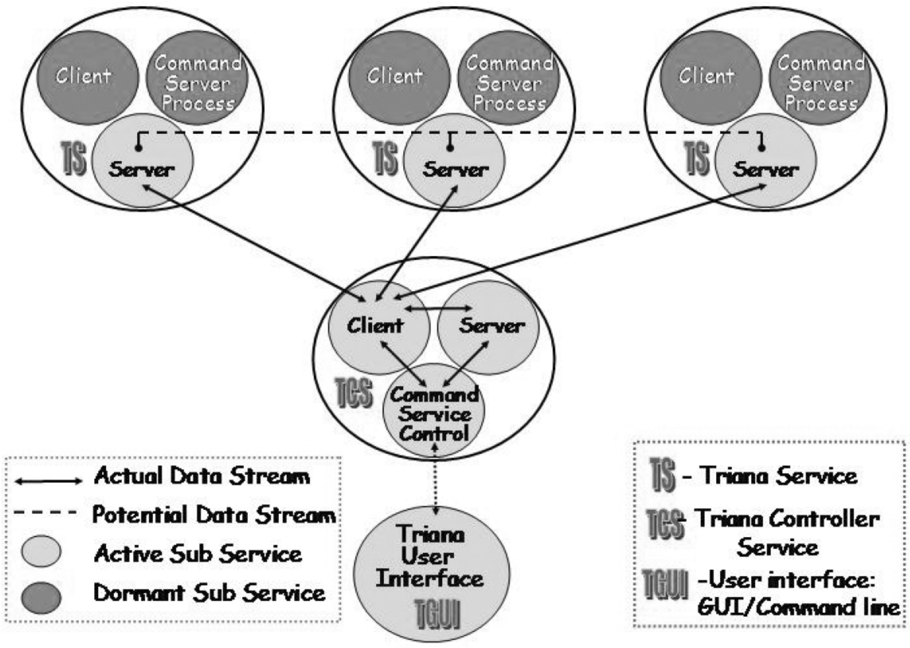


Figure 2.3 Triana architecture

Triana has three distinct components: a Triana Service (TS), a Triana Controller Service (TCS) and the Triana User Interface (TGUI). Figure 2.3 describes the interaction of these components. TGUI is a user interface to a TCS. The TGUI provides access to a network of computers running Triana service daemons via a gateway Triana service daemon and allows the user to describe the kind of functionality required of Triana: the module deployment model and data stream pathways. The TCS is a persistent service which controls the execution of the given Triana network. It can choose to run the network itself or distribute it to any available TS. Therefore, a single TGUI can control multiple Triana networks deployed over multiple CPU resources.

A Triana Service is comprised of three components: a client, a server and a command service controller. The client of the TCS in contact with the TGUI creates a distributed task graph that pipes modules, programs and data to the Triana service daemons. These Triana services simply act in server mode to execute the byte code and pipe data to others. Triana services can pass data to each other also. Clients (i.e. those running a GUI) can log into a Triana Controlling Service, remotely build and run a Triana network and visualize the result (using a graphical unit) on their device even though the visualization unit itself is run remotely. Users can also log off without stopping the execution of the network and then log in at a later stage to view the progress.

Triana communicates with services through the GAP Interface. The GAP Interface provides applications with methods for advertising, locating, and communicating with other services. The GAP Interface is, as its name suggests, only an interface, and therefore is not tied to any particular middleware implementation. This provides the obvious benefit that an application written using the GAP Interface can be deployed on any middleware for which there is a GAP binding. This also means that as new middleware becomes available, GAP based applications can seamlessly be migrated to operate in the new environment. Currently there are three middleware bindings implemented for the GAP Interface: JXTA, a peer-to-peer toolkit originally developed by Sun Microsystems; P2PS, a locally developed lightweight alternative to JXTA; and a Web services binding.

The Triana GUI supports different task graph writers. Workflows can be specified with Petri net, BPEL4WS [102] and a proprietary Triana XML format, and more can be added thanks to the pluggable architecture in Triana. This GUI supports looping constructs (e.g. do-while and repeat-until) and logic units (e.g. if, then etc.) that can be used to graphically control the dataflow, just as a programmer would control the flow within a program by writing instructions directly. In this sense, Triana is a graphical programming environment.

In addition to the individual tasks, tasks in the GUI can be also of two other types: group tasks and control tasks. Group tasks are composite tasks formed by the user by grouping one or more tasks within a workflow. They form the unit of distribution in Triana. If a user wishes to distribute a number of tasks then they create a group from those tasks, and designate that group for distribution through attaching a control task. Group tasks enable the user to determine the granularity of the workflow distribution and which tasks to distribute.

Control tasks are standard Triana units that are connected into the workflow to receive the input to a group before it is sent to the tasks within that group, and also to receive the output from a group before it is passed back to the main workflow. Control tasks have two roles: firstly, a control task is responsible for specifying the rewiring of the workflow to support some distribution topology (currently there are two supported: parallel and peer-to-peer) and secondly, as with looping control tasks, to redirect/manipulate the data input to/output from a group.

As a summary, Triana is a workflow engine with a very flexible architecture. In the upper layers it is intended to support different task graph writers, and new can be created to allow the possibility to specify the workflows in different formats. In the lower layers, the execution of the workflows can be done using different middlewares thanks to the GAP interface.

2.1.4 P-GRADE

P-GRADE [53] provides a high-level graphical environment to develop parallel applications transparently both for parallel systems and the Grid. One of the main advantages of P-GRADE is that the user does not have to learn the different APIs for parallel systems and the Grid. Simply by using the same environment will result in the generation of parallel applications transparently applicable either for supercomputers, clusters or the Grid. The current version of P-GRADE supports the interactive execution of parallel programs both on supercomputers, clusters and Globus Grids [33] as well as the creation of a Condor or Condor-G job to execute parallel programs as jobs in various (Globus and Condor) Grids.

Different kinds of Grids support different kinds of message passing programs. For example, Globus Grids support only the execution of MPI programs (they do not support PVM [39] program execution by default), while Condor provides more advanced parallel execution mechanism for PVM programs than for MPI [66] programs. Therefore, an important benefit of using P-GRADE is that the user can generate either PVM or MPI code according to the underlying Grid where the parallel application should be executed.

PVM applications generated by P-GRADE can migrate between different Grid sites and as a result P-GRADE guarantees reliable, fault-tolerant parallel program execution in the Grid, like the Condor system guarantees such features for sequential programs.

The GRM/PROVE [15] performance monitoring and visualization toolset has been extended towards the Grid and connected to a general Grid monitor (Mercury [14]) that was developed in the EU GridLab project. Using the Mercury/GRM/PROVE Grid application monitoring infrastructure any parallel application launched by P-GRADE can be remotely monitored and analyzed at run time even if the application migrates among Grid sites.

P-GRADE also supports workflow definition and coordinated multi-job execution for the Grid. Such workflow management supports parallel execution at both inter-job and intra-job level. Automatic checkpoint mechanism for parallel programs supports the migration of parallel jobs inside the workflow providing a fault-tolerant workflow execution mechanism. The same remote monitoring and performance visualization techniques used for the single parallel job execution mode can be applied for the workflow execution in the Grid. The current workflow engine is based on Condor DAGMan.

A P-GRADE workflow is an acyclic dependency graph that connects sequential and parallel programs into an interoperating set of jobs. The nodes of such a graph are batch jobs, while the edge connections define data relations among these jobs. Edges define the execution order of the jobs and the input/output dependencies that must be resolved by the workflow manager during execution. In the graphical interface, small squares labeled by numbers around the nodes are called ports and represent input and output data files that the corresponding executables expect or produce (one port represents one input/output file). Directed edges interconnect pairs of input and output ports if an output file of a job serves as an input file for another job. An input file can come from three different sources: it can be produced by another job, it can be in the workflow's developer desktop machine and it can be in a storage resource. Similarly, an output file can have three target locations: a computational resource (to be used by another job), the portal server (from where the user can download the file to his laptop), and a storage resource.

The P-GRADE Portal (a thin client concept of P-GRADE) is a workflow-oriented Grid portal with the main goal to support all stages of Grid workflow development and execution processes. It enables the graphical design of workflows created from various types of executable components (sequential, MPI or PVM jobs), executing these workflows in Globus-based computational Grids relying on user credentials, and finally, analyzing the monitored trace-data by the built-in visualization facilities. It gives a Globus-based implementation for the collaborative-Grid, collaborative-user concept. Since almost every production Grid uses Globus middleware today, these Grids could all be accessed by the P-GRADE Portal. Moreover, due to the multi-Grid concept a single portal installation can serve user communities of multiple Grids. These users can define workflows using the high-level graphical notations of the Workflow Editor, can manage certificates, workflows and jobs through the Web-based interface of the Portal Server. They can harness resources from multiple VOs and can migrate applications among Grids without learning new technologies or interfaces.

Figure 2.4 describes the structure of the P-GRADE Portal. One of the main features that makes the P-GRADE Portal unique from other portals or workflow environments is that it allows several users to work with the same workflow at the same time (this is named as multiple collaborative users). And also it allows to send parts of the workflow to different Grids from different VOs (what they call multiple collaborative Grids). So, it is an advanced portal where multiple users can work together to define and execute Grid applications that utilize resources of multiple Grids. By connecting previously separated Grids and previously isolated users together, these types of portals will revolutionize multidisciplinary research.

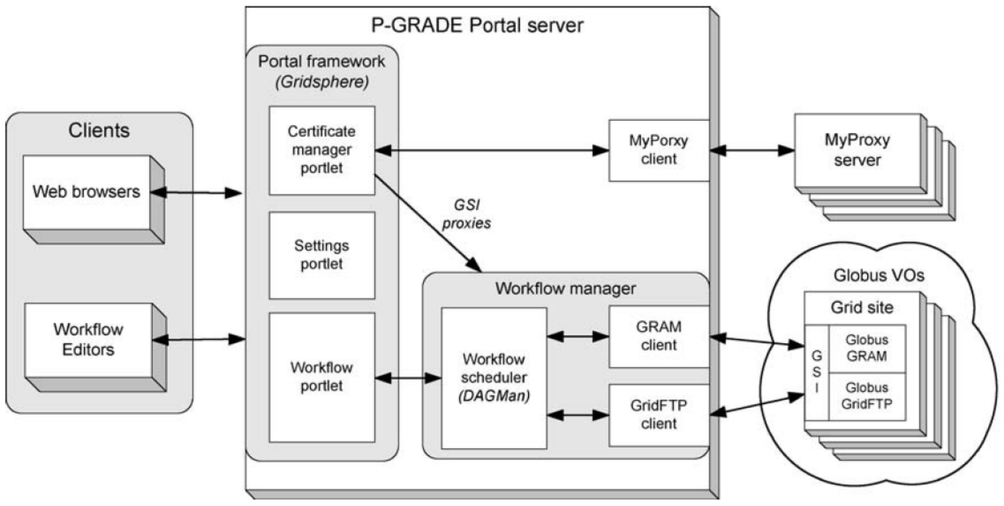


Figure 2.4 P-GRADE Portal structure

2.2 Programming models

2.2.1 Ninf-G

Ninf-G2 [97] is a client-server-based framework for programming on the Grid and a reference implementation of the GridRPC API [83] (a proposed OGF standard). Ninf-G2 has been designed so that it provides 1) high performance in a large-scale computational Grid, 2) the rich functionalities which are required to adapt to compensate for the heterogeneity and unreliability of a Grid environment, and 3) an API which supports easy development and execution of Grid applications. Ninf-G2 is implemented to work with basic Grid services, such as GSI, GRAM, and MDS in the Globus Toolkit version 2.

GridRPC is a programming model based on client-server-type remote procedure calls on the Grid. Client programs can "call" libraries on remote resources using the client API provided by GridRPC systems. A GridRPC system is required to provide a mechanism for "remote calls".

Two fundamental objects in the GridRPC model are function handles and the session IDs. The function handle represents a mapping from a function name to an instance of that function on a particular server. A session ID is used throughout the API to allow users to obtain the status of a submitted non-blocking call, to wait for a call to complete, to cancel a call, or to check the error code of a call.

GridRPC has several features that distinguish it from other RPC implementations. It provides scientific datatypes and IDL, e.g., large matrices and files as arguments, call-by-reference and shared-memory matrix arguments with sections/strides as part of a "Scientific IDL". GridRPC is very efficient in terms of bandwidth use since it does not

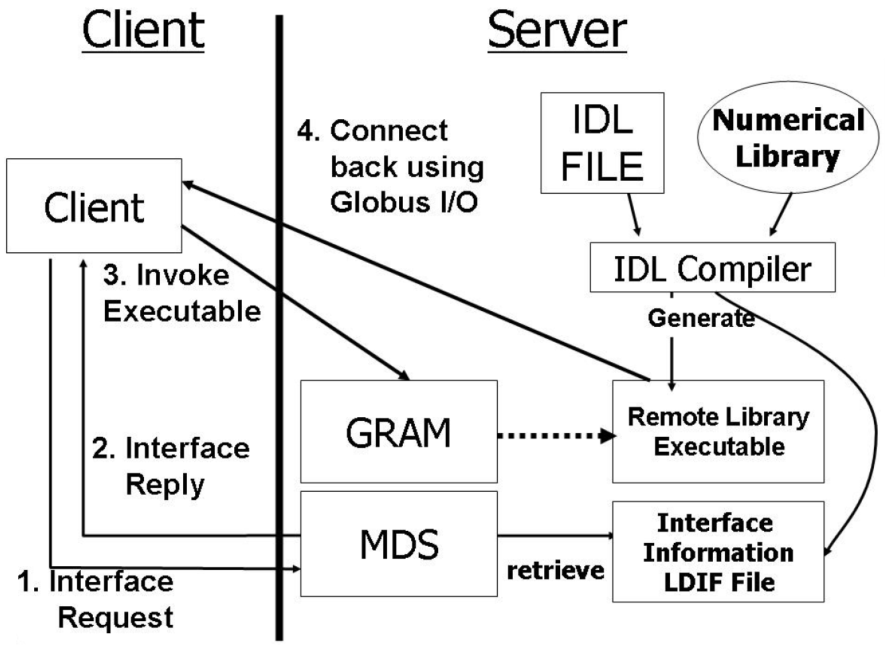


Figure 2.5 Ninf-G overview

send entire matrix when strides and array-sections are specified. In addition, it enables simple client-side programming and management, i.e., no client-side IDL management and very little state left on the client.

Ninf-G is a re-implementation of the Ninf system on top of the Globus Toolkit. The Client API has several functions that can be classified in different categories: Initializing and Finalizing; Function Handle Management (create, destroy, ...); GridRPC Call (blocking or non-blocking and it may use variable number of arguments or an argument stack calling sequence); Asynchronous GridRPC Control (for non-blocking requests); Asynchronous GridRPC Wait; Error Reporting Functions (human-readable error descriptions; Argument Stack Functions (to construct the arguments for a function call at run time).

In order to gridify and provide libraries and applications using Ninf-G, the provider is required to follow a simple few steps: (1) Describe the interface for the library function or an application with Ninf IDL, (2) compile the description and generate a stub main routine and a makefile for the remote executable, (3) compile the stub main routine and link it with the remote library, (4) publish information on the remote executable. Steps (3) and (4) are automated by the makefile. An overview of these steps is described in Figure 2.5.

Ninf IDL (Interface Description Language) is used to describe interface information for gridified libraries. In addition to the interface definition of the library function, the IDL description contains the information needed to compile and link the necessary libraries. Ninf-G tools allow the IDL files to be compiled into stub main routines and makefiles, which automates compilation, linkage, and registration of gridified executables.

The runtime included in Ninf-G does not provide fault recovery, or load-balancing by itself. Ninf-G propagates an appropriate error status to the client if the error occurs at either the client or the server side, however Ninf-G itself does not try to recover from the errors. Instead, Ninf-G assumes that a back-end queuing system, such as Condor [101], takes responsibility for these functions. Otherwise, application programmers should handle the errors by themselves.

An important thing to mention is that the host name is explicitly specified as part of the API, i.e. the host name must be given when a function handle is initialized. Ninf-G provides no scheduling feature, however it is possible and easy to write master-worker-type self-scheduling-based applications using simultaneous asynchronous calls through multiple function handles provided by Ninf-G.

Ninf-G employs the following components from the Globus toolkit [33]: GRAM, invokes remote executables. MDS (Monitoring and Discovery Service) publishes interface information and pathnames of GridRPC components. Globus I/O is used for the communication between clients and remote executables. GASS redirects stdout and stderr of the GridRPC component to the client tty.

There is also a Java client API available. The availability of the Java client has several advantages for developing Ninf-G-based applications. The implementation is built on the top of the Java Commodity Grid Toolkit (CoG Kit) [108]. The Java CoG Kit enables access for a variety of Globus services and facilitates the development of all protocols between Ninf-G Java clients and Globus services. The Java Client API provides no asynchronous API, unlike the C API because the Java language itself supports multi-threaded programming.

The latest developments implemented in Ninf-G are a Task Farming API and a Task Sequencing feature. Task Farming runs the same program in parallel while changing input data and parameters. The Task Farming API enables programmers to produce task farming code easily, and to have almost the best performance and stability possible without a great amount of effort. In Task Sequencing, RPC requires dependency between input and output parameters, which means output of a previous RPC becomes the input of the next RPC. In this work, the direct transfer of data between servers (not supported in the GridRPC standard) is implemented using a Grid filesystem without destroying the GridRPC programming model and without changing very many parts of the existing Ninf-G implementation.

2.2.2 Satin

The Satin programming model [105] has been designed specifically for running divide-and-conquer programs on distributed memory systems and wide-area Grid computing systems. In Satin, single-threaded Java programs are parallelized by annotating methods that can run in parallel. The ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area systems, and more general, on the Grid. The divide-and-conquer model maps efficiently on such systems, as the model is also hierarchical. Satin is based on the Manta [63] native compiler, which supports highly efficient serialization and communication. Parallelism is achieved in Satin by running different spawned method invocations on different machines. The system load is balanced by the Satin runtime system.

Satin's programming model is an extension of the single-threaded Java model. Satin programmers thus need not use Java's multithreading and synchronization constructs or Java's Remote Method Invocation mechanism, but can use the much simpler divide-and-conquer primitives described below.

Parallel divide-and-conquer systems have at least two primitives: one to spawn work, and one to wait until the spawned work is finished. Satin exploits Java's standard mechanisms, such as inheritance and the use of marker interfaces (e.g., java.io.Serializable) to extend Java with divide-and-conquer primitives.

In Satin, a spawn operation is a special form of a method invocation. Methods that can be spawned are defined in Satin by tagging methods with a special marker interface. These methods are called Satin methods. A call to a method that was tagged as spawnable is called a spawned method invocation. With a spawn operation, conceptually a new thread is started which will run the method (the implementation of Satin, however, eliminates thread creation altogether). The spawned method will run concurrently with the method that executed the spawn. The sync operation waits until all spawned calls in this method invocation are finished. The return values of spawned method invocations are undefined until a sync is reached. The assignment of the return values is not guaranteed to happen in the sync operation, but instead is done between the spawn and sync. All operations that exit a method (e.g., return or throw) are treated as an implicit sync operation. Spawned methods can throw exceptions, just like ordinary Java methods.

Because spawned method invocations may run on any (remote) machine, global variables that are defined on the machine that spawned the work may not be available at the site where the method is executed. The code of the spawned method should only access data that is stored on the machine that runs the work. Essentially, this means that Satin methods must not change global state. In pure Satin, the only way of communicating between jobs is via the parameters and the return value. In short, Satin methods must not have side-effects. However, Satin programmers can use Manta's replicated object system (RepMI) to implement shared data.

When a program executes a spawn operation, Satin redirects the method call to a stub. This stub creates an invocation record, describing the method to be invoked, the parameters that are passed to the method, and a reference to where the method's return value (or exception) has to be stored. For primitive types, the value of the parameter is copied. For reference types (objects, arrays, interfaces), only a reference is stored in the record.

The Java's serialization mechanism is used in Satin for marshalling the parameters to a spawned method invocation. Satin implements serialization on demand: the parameters are serialized only when the work is actually stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. When an object has reference fields, the serializers for the referenced objects will also be called, so all data that can be reached (directly or indirectly) via those parameters is serialized.

The invocation records describing the spawned method invocations are stored in a double-ended job queue. Newly generated work is always inserted at the head of the queue. Locally, work is always taken from the head of the queue, thus the queue is effectively used as a stack. When a node runs out of work, it will start stealing work from other nodes. Idle nodes will poll remote queues for jobs, at the tail of the queue. The reason why local nodes execute work from the head of the queue, while remote nodes steal at the tail of the queue is that in this way large-grain jobs are stolen, reducing communication overhead.

Java's garbage collector can be a problem to Satin runtime (implemented in C), as spawns are asynchronous (the method that executed the spawn continues) and the garbage collector could assume that object parameters passed to a spawned method invocation are no longer in use. Satin registers the invocation records at the garbage collector to avoid this problem.

Five load-balancing algorithms are implemented in the Satin runtime system to investigate their behavior on hierarchical wide-area systems. Random Stealing (RS) (good for single-cluster environments), hierarchical load balancing (proposed by other groups), Cluster-aware Random Stealing (CRS) (achieves good speedups for a large range of bandwidths and latencies), Random Pushing and Cluster-aware Load-based Stealing (a variant of hierarchical stealing).

The latest developments in Satin include adding fault tolerance mechanisms by using a global result table to minimize the amount of redundant computation.

2.2.3 ProActive

ProActive [13] is an Open Source Java library for parallel, distributed, and multi-threaded computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive toolkit that simplifies the programming of applications distributed on Local Area Networks (LANs), Clusters, Internet Grids and Peer-to-Peer Intranets. ProActive is the first framework featuring hierarchical distributed components.

As ProActive is built on top of the Java standard APIs, mainly Java RMI and the Reflection APIs, it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine. Additionally, the Java platform provides dynamic code loading facilities, very useful for tackling with complex deployment scenarios.

Active objects are the basic units of activity and distribution used for building concurrent applications using ProActive. As opposed to passive (regular) objects, the active object has its own thread and execution queue. ProActive manages the active objects threads relieving the programmer from explicitly manipulating Thread objects thus making the use of the threads transparent. We can see an example of an object graph with active objects in Figure 2.6.

When an object in a subsystem calls a method on an active object, the parameters of the call may be references on passive objects of the subsystem, which would lead to shared passive objects. This is why passive objects passed as parameters of calls on active objects are always passed by deep-copy. Active objects, on the other hand, are always passed by reference. Symmetrically, this also applies to objects returned from methods called on active objects.

When a method is called on an active object, it returns immediately (as the thread cannot execute methods in the other subsystem). A future object, which is a placeholder for the result of the methods invocation, is returned. From the point of view of the caller

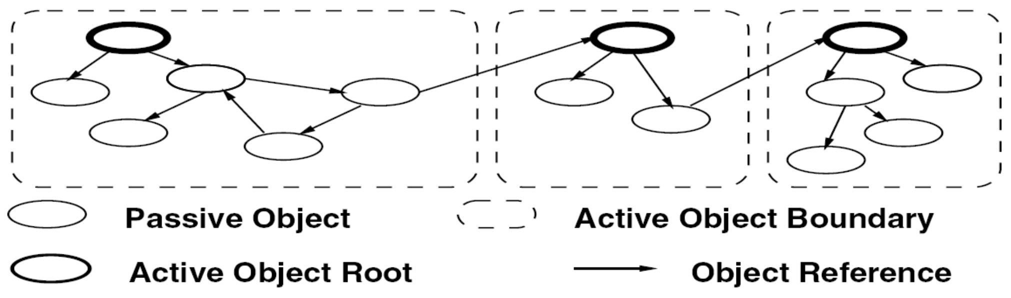


Figure 2.6 Object graph with active objects

subsystem, no difference can be made between the future object and the object that would have been returned if the same call had been issued onto a passive object. Then, the calling thread can continue executing its code just like if the call had been effectively performed. The role of the future object is to block this thread if it invokes a method on the future object and the result has not yet been set (i.e. the thread of the subsystem on which the call was received has not yet performed the call and placed the result into the future object). This type of inter-object synchronization policy is known as wait-by-necessity.

Active objects have the ability to migrate between nodes while the application is running. Whatever application was using the active objects will continue running regardless of the migration of active objects. The migration of an active object can be triggered by the active object itself, or by an external agent. In both cases a single method will eventually get called to perform the migration (the method migrateTo(...)).

In order to ensure the correct execution of an application in the presence of migration, three mechanism are provided to maintain communication with mobile objects. The first one relies on a location server which keeps track of the mobile objects in the system. When needed, the server is queried to obtain an up-to-date reference to an active object. After migrating, an object updates its new location. The second one uses a fully decentralized technique known as forwarders. When leaving a site, an active object leaves a special object called a forwarder which points to its new location. Upon receiving a message, a forwarder simply passes it to the object (or another forwarder). The third one is an original scheme based on a mix between forwarding an location server which provides both performance and fault tolerance.

The group communication mechanism of ProActive achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies. A method invocation on a group has a syntax similar to that of a standard method invocation. Such a call is asynchronously propagated to all members of the group using multi-threading.

The effective locations of active objects are not indicated in the source code thanks to the deployment descriptors. Parameters tied to the deployment of an application are described in an XML deployment descriptor. Hence within the source code, there are no longer any references to: machine names, creation protocols, registry/lookup and communications protocols or file transfers.

The aim of the ProActive work around components is to combine the benefits of a component model with the features of ProActive. The resulting components, called "Grid components", are recursively formed of either sequential, parallel and/or distributed sub-components, that may wrap legacy code if needed, and that may be deployed but further reconfigured and moved, for example to tackle fault-tolerance, load-balancing or adaptability to changing environmental conditions.

The Architecture Description Language (ADL) is used to configure and deploy component systems. The architecture of the system is described in a normalized XML file. The ADL automates the instantiation, the deployment, the assembly and the binding of the components.

The ProActive framework also includes a batch scheduler, which provides an abstraction of resources to users. Users submit jobs containing tasks to the scheduler, who is in charge of executing these tasks on the resources. A scheduler allows several users to share a same pool of resources and also to manage all issues related to distributed environment, such as faulted resources. By default, the scheduler schedules tasks according to the default FIFO with job priority policy, but the policy can be changed easily.

IC2D is a graphical environment included in ProActive for remote monitoring and steering of distributed and grid applications. IC2D is available in two forms: A Java standalone application and a set of Eclipse plugins. The also included TimIt API offers a complete solution to benchmark an application.

2.2.4 CoG Kits

A Commodity Grid Toolkit (CoG Kit) defines and implements a set of general components that map Grid functionality into a commodity environment/framework. CoG Kits provide a mapping between computing languages, frameworks, and environments and Grid services and fabric. Together Grid services, languages, frameworks, and environments build a powerful development tool for building grid-enhanced applications. Hence, we can imagine a Web/CGI CoG Kit, a Java CoG Kit, a CORBA CoG Kit, a DCOM CoG Kit, and so on. In each case, the benefit of the CoG Kit is that it enables application developers to exploit advanced Grid services (resource management, security, resource discovery) while developing higher-level components in terms of the familiar and powerful application development frameworks provided by commodity technologies. We can see where those mappings must be applied in Figure 2.7. The main goal among all possible CoG Kits has been to build a Java CoG Kit.

The Java CoG Kit [108] leverages the portability and availability of a wide range of libraries associated with the Java framework, while promoting easy and rapid Grid application development. The Java CoG Kit started with the development of a client-side and partial server-side implementation of the classic Globus (Globus Toolkit 2.x) libraries under the name of "jglobus." Today jglobus includes, among other libraries, Java

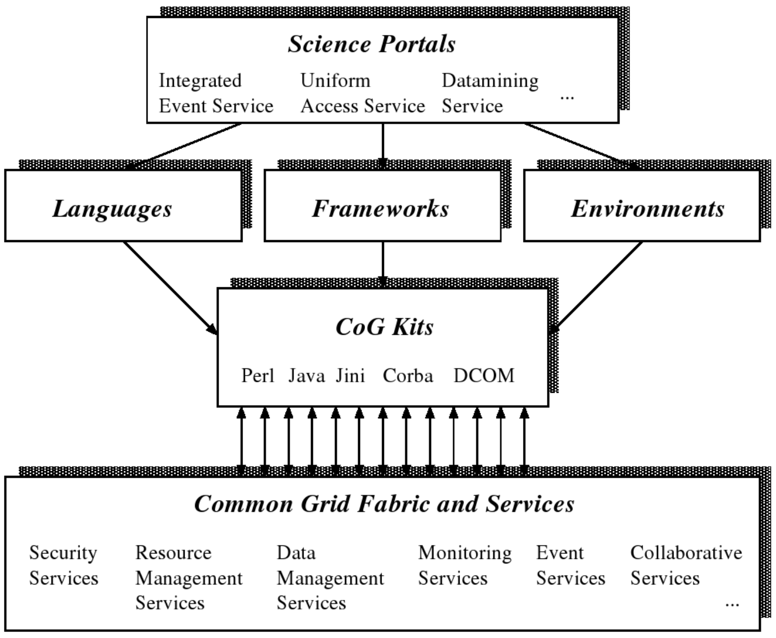


Figure 2.7 CoG Kits mapping objective

implementations of the Grid Security Infrastructure (GSI) libraries, GridFTP, myProxy, and GRAM. The jglobus library is a core component of both Globus Toolkit 3 and Globus Toolkit 4 and a major contribution of the Java CoG Kit to the Grid effort.

One of the concepts that has proven to be useful in protecting the user from frequent changes in the standards development is the concept of abstractions and providers. Through simple abstractions, they built a layer on top of the Grid middleware that satisfies many users by giving them access to functions such as file transfer or job submission. These functions hide much of the internal complexity present within the Grid middleware. Furthermore, it projects the ability to reuse commodity protocols and services for process execution and file transfer instead of only relying on Grid protocols. In addition to the abstraction and provider concept, the Java CoG Kit also provides user-friendly graphical tools, workflows, and support for portal developers. End users will be able to access the Grid through standalone applications, a desktop, or a portal. Command-line tools allow users to define workflow scripts easily. Programming is achieved through services, abstractions, APIs, and workflows.

As we have seen, the toolkit is based on the abstractions and providers model. They identified a number of useful basic and advanced abstractions that help in the development of Grid applications. These abstractions include job executions, file transfers, workflow abstractions, and job queues and can be used by higher-level abstractions for rapid prototyping. As the Java CoG Kit is extensible, users can include their own abstractions and enhance its functionality. The concept of Grid providers allows a variety of Grid middleware to be integrated into the Java CoG Kit. Components in the Java CoG Kit are classified in four categories: Low-Level Grid Interface Components, Low-Level Utility Components, Common Low-Level GUI Components and Application-specific GUI Components.

Low-Level Grid Interface Components provide mappings to commonly used Grid services: for example, the Grid information service (the Globus Metacomputing Directory Service, MDS), which provides Lightweight Directory Access Protocol (LDAP) access to information about the structure and state of Grid resources and services; resource management services, which support the allocation and management of computational and other resources (via the Globus GRAM and DUROC services); and data access services, for example, via the Globus GASS service.

Low-Level Utility Components are utility functions designed to be reused by many users. Examples are components that use information service functions to find all compute resources that a user can submit to; that prepare and validate a job specification while using the extended markup language (XML) or the Globus job submission language (RSL); that locate the geographical coordinates of a compute resource; or that test whether a machine is alive.

Common Low-Level GUI Components provide a set of low-level GUI components that can be reused by application developers. These components provide basic graphical components that can be used to build more advanced GUI-based applications. They include text panels that format RSL strings, tables that display results of MDS search queries, trees that display the directory information tree of the MDS, and tables to display Heart Beat Monitor (HBM) and network performance data.

Application-specific GUI Components simplify the bridge between applications and the basic CoG Kit components. Examples are a stockmarket monitor, a graphical climate data display component, or a specialized search engine for climate data. High-level graphical applications combine a variety of CoG Kit components to deliver a single application or applet. Naturally, these applications can be combined in order to provide even greater functionality. The user should select the tools that seem appropriate for the task.

The latest development in the Java CoG kit is a workflow framework called Karajan [109]. This framework can be used to specify workflows through a sophisticated XML scripting language as well as an equivalent more user-friendly language termed K. In contrast to other systems it not only supports hierarchical workflows based on DAGs but also have the ability to use control structures such as if, while, and parallel in order to express easy concurrency. The language itself is extensible through defining elements, and through simple data structures it allows easy specification of parameter studies. The workflows can be visualized through a simple visualization engine, which also allows monitoring of state changes of the workflow in real time. The workflows can actually be modified during runtime with a workflow repository that gets called during runtime or by the specification of schedulers that allow the dynamic association of resources to tasks (because the binding of tasks to resources and protocols can either be done explicitly or delegated to a scheduler).

2.2.5 Swift

Swift [114] is a system that combines a novel scripting language called SwiftScript with a powerful runtime system based on CoG Karajan (see section 2.2.4) and Falkon [76] to allow for the concise specification, and reliable and efficient execution, of large loosely coupled computations. Swift adopts and adapts ideas first explored in the GriPhyN virtual data system, improving on that system in many regards. It provides the XML Dataset Typing and Mapping [65] model for clean separation of logical data structures and physical storage formats, a scripting language SwiftScript for concise specification and convenient composition of large complex workflows, and a scalable runtime system that manage and dispatch hundreds of thousands parallel computations, onto a variety of parallel and distributed computation facilities. The Swift system provides a wide range of capabilities to support the formulation, execution and management of large compute- and data-intensive computations.

XDTM allows logical datasets to be defined in a manner that is independent of the datasets' concrete physical representations. XDTM employs a two-level description of datasets, defining separately via a type system based on XML Schema the abstract structure of datasets, and the mapping of that abstract data structure to physical representations.

A dataset's logical structure is specified via a subset of XML Schema, which defines primitive scalar data types such as Boolean, Integer, String, Float, and Date, and also allows for the definition of complex types via the composition of simple and complex types. The use of XML Schema as a type system has the benefit of supporting powerful standardized query languages such as XPath in the selection methods.

A dataset's physical representation is then defined by a mapping descriptor, which describes how each element in the dataset's logical schema is stored in/fetched from physical structures such as directories, files, and database tables. In order to permit reuse for different datasets, mapping descriptors typically refer to external parameters for such things as dataset locations.

SwiftScript allows concise definitions of logical data structures and logical procedures that operate on them, and complex computations to be composed from simple and compound procedures. Its support for nested iterations can allow a compact SwiftScript program to express hundreds of thousands of parallel tasks. SwiftScript programs can be at least an order of magnitude smaller in lines of code than other approaches such as Shell scripts and directed acyclic graph specifications.

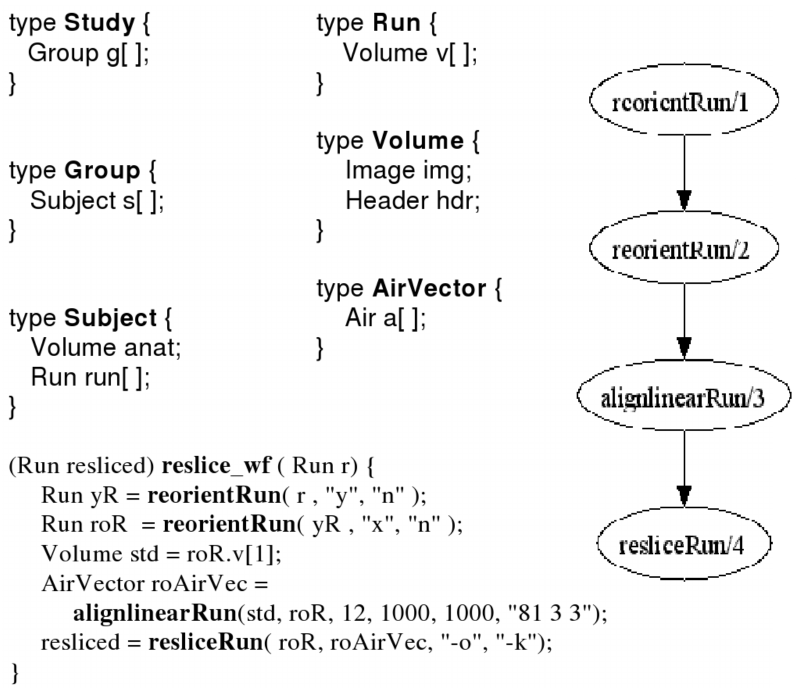
The SwiftScript scripting language builds on XDTM to allow for the definition of typed data structures and procedures that operate on such data structures. SwiftScript procedures define logical data types and operations on those logical types; the SwiftScript implementation uses mappers to access the corresponding physical data. In addition to providing the usual advantages of strong typing (type checking, selfdocumenting code, etc.), this approach allows SwiftScript programs to express opportunities for parallel execution easily, for example by applying transformations to each component of a hierarchically defined logical structure. Figure 2.8 contains an example of a SwiftScript program as well as the dependencies that would be generated from it. The body of a procedure can include an invocation of an executable program (or a Web Service). That program will be dynamically mapped to a binary executable, which will be executed at an execution site chosen by the Swift runtime system. The body may also specify how input parameters map to command line arguments (i.e. the notation @filename is a built-in mapping function that maps a logical data structure to a physical file name).

The Swift runtime system is a scalable environment for efficient specification, scheduling, monitoring and tracking of SwiftScript programs. For program specification, computations are defined in SwiftScript programs, that are compiled by a SwiftScript compiler into abstract computation plans, which can be scheduled for execution by the execution engine.

In order to schedule programs, Swift uses CoG Karajan as its underlying execution engine. Karajan provides a set of convenience libraries for file operation, data transfer, job submission, and Grid services access. Karajan has been extended with specific libraries to support the XDTM type system and logical dataset manipulation, adapters to access legacy VDS components (for instance, site catalog), mappers for accessing heterogeneous physical data storage, and also with fault tolerance mechanisms.

For executing a program, intermediate abstract execution plans represented in customized Karajan scripts are interpreted and dispatched by Karajan onto execution sites. Swift determines at runtime the actual execution processes such as site selection, data stage-ins and stage-outs, and error checking. A set of callouts allow customized functions to determine where to dispatch tasks, how to group tasks to increase granularity, and/or when and how to perform data staging operations etc. Swift also supports advanced Grid scheduling and optimization, such as load balance, fault tolerance, computation restart, etc. Individual tasks are invoked by a launcher program (for instance, kickstart) which monitors and records the execution process and gathers provenance information.

In brief, Swift can execute large-scale computations in both parallel computers and distributed systems. It focuses on scripting as the preferred language for scientists by providing SwiftScript. Logical data structures can be declared as well as typed procedures to iterate over such datasets, including a nested approach. Users work at a clean abstract level thanks to the dataset mappings, that allows the system to know how to access or transport datasets. Both service-oriented and command-oriented applications can be executed with the XDTM model. The combination of such distinctive features enables the automation of scientific data and workflow managements, and improves usability and productivity in scientific applications and data analyses.



```
type Study {
  Group g[ ];
}

type Group {
  Subject s[ ];
}

type Subject {
  Volume anat;
  Run run[ ];
}

type Run {
  Volume v[ ];
}

type Volume {
  Image img;
  Header hdr;
}

type AirVector {
  Air a[ ];
}

(Run resliced) reslice_wf ( Run r) {
   Run yR = reorientRun( r , "y", "n" );
   Run roR  = reorientRun( yR , "x", "n" );
   Volume std = roR.v[1];
   AirVector roAirVec =
      alignlinearRun(std, roR, 12, 1000, 1000, "81 3 3");
   resliced = resliceRun( roR, roAirVec, "-o", "-k");
}
```

Figure 2.8 SwiftScript program and dependency graph example

2.3 Comparison with GRIDSs

DAGman is a utility for Condor to create executions with workflows. However, we have seen that its focus is at a very low level, as every dependency between two tasks must be specified manually by the end user. Moreover, the data transfers need also to be detailed by users. These two characteristics makes essential the usage of higher level tools for end users, as Pegasus.

Pegasus is a powerful tool for creating workflows to be executed in the Grid. Its locality policy is very similar to the one included in GRIDSs, and includes advanced features to avoid repeating calculations for different users. However, users must publish their data in the replica system in order to exploit its features, which may not be easy for them. Also a good knowledge of the Chimera's VDL is needed from users in order to ease the task of writing the definition of the workflow and no features to avoid concurrency problems are included (such as the parameter renaming in GRIDSs). Another drawback is that its implementation is completely dependant on Globus services, in contrast to GRIDSs, that is able to use other middlewares.

Triana is a flexible PSE that allows a graphical way of programming the execution of an application. It offers a flexible architecture where workflows can be specified in different standard formats (and new ones can be included), and the framework can be easily ported to different middlewares thanks to a standard interface (thus, only a binding must be developed to port it to a new middleware). However, data movement and Grid job submission all need to be explicitly specified and organized by end users in the control tasks, and this may be difficult for them.

The P-GRADE Portal is based on a web interface, which removes the need of installing software to access the Grid. It supports multiple users working in the same workflow, and to execute parts of a workflow in multiple Grids. Moreover, a job can be defined as a parallel tasks in the workflow (a task that uses PVM or MPI). Anyway, the user needs to draw in the interface exactly what he needs to execute, specifying all input/output files related to a task, and connecting explicitly two tasks when there is a data dependence between them. This limits the scalability of the system, as no loops or if clauses can be defined graphically, and it is very difficult for a user to build a complex graphical workflow with thousands or millions of nodes. Another important drawback is that it is explicitly developed to work with the Globus middleware and the DAGMan meta-scheduler in order to run the workflows.

We found some similarities between GRIDSs and Ninf-G: they both are based in the master-worker paradigm and they include an IDL to describe the interfaces of what the user wants to execute in the Grid (providing tools that generate intermediate code from that IDL to enable the remote execution). However, the API they provide is big enough to be considered difficult to be learnt by a non-expert user in computer science. Moreover, any parallelism must be explicitly built by the user, and even no scheduling is included (i.e. where a job is going to be run must be specified by the user). The latest developments in Ninf-G are features similar to the ones already supported in GRIDSs (parameter sweep and data dependency treatment). Their support of different programming languages is limited to C and Java, and it is developed as completely independent versions, in contrast to GRIDSs, that only a binding is developed to support a new programming language. We consider Ninf-G a tool applied at a lower level than GRIDSs.

The Satin programming model is focused on Java implemented divide-and-conquer applications, where the algorithm can be expressed recursively. This type of applications is also supported by GRIDSs and it is not only restricted to divide-and-conquer algorithms. Satin's focus to Java allows to implement passing complex parameters to remote executions by using serialization. Also they include diverse scheduling policies suitable for different environments. One added difficulty for users is that they need to specify when parallelism can be used with calls to spawn and sync methods, in contrast to GRIDSs, where parallelism is extracted automatically from the code.

As a powerful framework, ProActive offers a good solution in order to program parallel applications in the Grid. Its interface, however, may be too complicated to learn for general scientists, because of its big set of primitives. They try to reduce this gap by providing templates for some generic algorithms (Monte Carlo, ...). Also their focus is on Java applications, and applications implemented with other languages can be ported to ProActive with a call to a binary using a wrapper technology. The wait-by-necessity mechanism is somehow similar to the data dependence analysis provided in GRIDSs, by allowing data to flow through future objects, but no feature to remove data dependencies is included, as in GRIDSs. The parallelism inside an active object is easily achieved by asynchronous method calls, but parallelism between different active objects may be created by using several active objects at the same time, which is more complicated to implement for an end user.

The CoG Kits try to bring together the commodity and the Grid technologies by simplifying the development of the latter. With this extra layer, developers are protected from changing details in the underlying Grid middleware. The main problem is that the user who programs something on top of a CoG Kit must be a Grid expert, and not a general scientist, because a lot of details from the underlying Grid technology are still visible at this level (MDS, RSL, etc.), thus not making the Grid invisible. Karajan adds the workflow abstraction to the Java CoG Kit in such a way that users must generate the workflow in a particular language provided by them that users must learn from scratch (thus making more difficult its adoption).

The Swift approach is based in scripting as a main language. This is similar to what GRIDSs is based on (imperative languages) and has the main advantage of defining easily complex workflows by generating them with the language, but with two main drawbacks. The first one is the need of learning this new language (SwiftScript) for new users, which may not be as easy as it may seem, while GRIDSs can be used with already known programming languages. And the second one is the lack of possibility of calculating anything inside the script file: with GRIDSs you can decide which parts of the code you want to execute in the Grid, but SwiftScript is only capable of defining how to execute external solvers, and nothing can be solved inside the program. Swift main focus is to iterate over datasets in an easy way, but it is somehow surprising that no policy related to exploiting any singularities in the data is included in their scheduler, as it is in GRIDSs. Moreover, nothing similar to parameter renaming is provided in order to avoid concurrency problems between tasks that can run in parallel.

2.4 Fault tolerance in Grid environments

The fault tolerance in the Grid is applied at different levels in already existing work. The detection and recovery of failures is done at a Grid level (Grid middleware and host errors) or at the application level. The main advantage in implementing fault tolerance features at the application level is that both Grid level failures and application errors can be handled.

2.4.1 Grid level

According to [46] we can distinguish between push and pull methods to detect failures in a resource. The push method is based in heartbeat messages sent from remote resources. The pull method works by polling for the state of the resources monitored. If a reply message is not received before a specified timeout, the resource is considered to be temporary unavailable.

We can see an example of the push method in Stelling et. al fault detection service [92], which can provide fault detection capabilities to middlewares without them. The pull method has been implemented in the GridBus [107] and GridWay [49] meta-schedulers. Both push and pull methods are unable to determine if the failure is in the network or the host.

In contrast, Phoenix [57] can detect failures scanning the scheduler log files, and the way these failures are corrected can be defined by the user. However, if the system does not have log files that can be interpreted, Phoenix cannot be applied.

In [104], the authors present the Java Grid Application Toolkit (JavaGAT) that provides a high-level middleware-independent and site-independent interface to the Grid. This API implements *intelligent dispatching* that can automatically select the best middleware for each individual Grid operation. When a particular middleware fails, the JavaGAT engine automatically selects alternative implementations for the requested operation until one succeeds, providing transparent fault tolerance, and solving heterogeneity problems. In case all available middleware implementations fail to provide a requested operation, JavaGAT provides the application with a special, nested exception, enabling detailed error analysis when needed. The overhead of using intelligent dispatching is insignificant compared to the cost of the Grid operation. Therefore, the use of the JavaGAT enables fault tolerance at the middleware level, but not a complete level of fault tolerance.

2.4.2 Application level

Fault tolerance at the application level is usually implemented directly into the application code. One of the most important things is that, at this level, not only Grid level failures but also application specific errors can be treated and recovered. This approach can provide flexibility and efficiency by exploiting application-level knowledge but it is quite difficult for the application programmers [64].

Most approaches in the literature to ensure fault tolerance at the application level are based on checkpoint recovery and replication [111]. Checkpointing is a widely used technique, that consists of saving the state of the application at given points in time and then rollback towards these safe states in case of failure.

The Open Grid Forum [70] has a Working Group dedicated to Checkpoint and Recovery (GridCPR-WG). They have produced two documents regarding this subject ([6] and [95]) but their work is still far from achieving a standard for checkpoint and recovery techniques, and it has been outdated with new proposals for Grid architectures. This may be the reason why in the literature we can find many different and tailored approaches, rather than following a common standard.

In EU project DataGrid [41], an API for the checkpointing service is offered. The user must define what the state of a job is (with variable/value pairs). These set of pairs have to be enough to restart a computation from a previously saved state. Users can save the state of the job, and the first instruction of their code should be the retrieval of the last saved state. The restart can be from any previous saved state, not necessarily the last one. The checkpoint is saved in the Logging and Bookkeeping server.

The Pittsburgh Supercomputing Center CPR [94] was originally designed for parallel jobs, not for serial job "farms". A job can be restarted in a machine if that machine can read the checkpoint file (thus, there exists heterogeneity problems). It includes checkpoint file recovery schema, checkpoint file migration service, scheduling integration and pre-emption requests.

In C3 [20], an strategy oriented to provide transparent application-level checkpointing to MPI applications by means of a pre-processor is presented. It is considered that a sequential checkpoint can be done saving global variables, stack frames and heap allocated objects. It is considered that sequential checkpointing is sufficient to do parallel checkpointing in parametric computing.

Cactus [42] uses a strategy that allows checkpointing and restart of thorns. The checkpoint is oriented to the application thorns that form an entire simulation. All the basic simulation data is well known, so they checkpoint simulation data and the parameters to recreate this data. Checkpointing may happen after the calculation of initial data, periodically during the simulation, at user request or at the end of the simulation. Thorn developers need to check that their thorns can be checkpointed and restarted. As it is very tailored to Cactus, it is not a general checkpoint mechanism.

In RealityGrid project [78] the checkpoint is written by the application (it has to register each checkpoint with the library and with a tag to identify it). This checkpoint can be done for simulation components (typically a single parallel program), but not for other components (as visualization ones). It is considered that the checkpoint must allow restarting a job which was initially running on N processors on a new M processors configuration. The main focus as future work also includes windbacks (revert to the state captured in a previous checkpoint without stopping the application). But the final goal is to achieve computational steering (interact and change the behavior of a running application) by means of checkpoint tree structures.

In Condor [101] the checkpointing mechanism saves all the state of a process into a checkpoint file. The process can then be restarted from right where it left off. Typically, no changes to the job's source code are needed. When Condor sends a checkpoint signal to a process, its state is written out to a file or a network socket. This state includes the process stack, data segments, all shared library code and data mapped into the process's address space, the state of all open files and any signal handlers and pending signals. It has some limitations: the Condor checkpointing code must be linked in with the user's code, the code cannot call fork, use kernel threads or use some forms of IPC (pipes and shared memory). It has been incorporated in commercial products such as LoadLeveler.

LoadLeveler [55] supports checkpointing of running serial and parallel jobs. When a job becomes checkpointed, a memory dump of the job's processes and message queues is written into one or more files. There is a list about its limitations: MPI jobs that are not linked with the threaded communication libraries, jobs that use Large-Memory pages, parallel jobs cannot be restarted if the number of tasks or the task geometry differs from the state stored in the checkpoint file, is problematic when jobs use shared memory, and so on.

In GridLab [82], the GRMS (GridLab Resource Management System) supports checkpointing for applications which are able to register callback information in the GRMS (so, they can wait for a checkpoint GRMS's call). It is an application-level checkpoint in which the application is forced to store all information required for restart in one or more checkpoint files.

In [112], the authors propose a fault tolerance mechanism for SATIN. SATIN is a programming model for divide-and-conquer applications build on top of Ibis. In their paper, the authors propose a technique based on storing partial results in a global table. These results can be later reused, therefore minimizing the amount of work lost as a results of a crash.

In [58], a Reliable Execution Service (RES) is presented, which is defined as a persistently running service that keeps track of an application's life cycle. Such a service is supposed to become and integral part of a Grid application execution environment. The RES ensures a reliable execution of applications submitted to the Grid. The RES enables to users to take care of the application errors only, with the exception of the permanent Grid failures. On failure, the service classifies it on permanent or transient. Permanent failures are reported to the user and transient failures are shielded through recovery techniques, basically by retrying the execution on a different resource. Although it is closer to the application level, it might be more difficult to apply when complex applications are to be run (such as workflows). The paper also presents a case study with GRIDSs.

Task replication consists of submitting redundant instances of the tasks to increase the probability of having one instance successfully executed. In [61] the authors present an approach for fault tolerance in Mobile Grid environments by means of task replication. The approach is based on static task replication using the Weibull reliability function for the Grid resources so as to estimate the number of replicas that need to be scheduled to guarantee a specific fault tolerance level (i.e., that a given probability of task completion is ensured). These replicas are then scheduled by means of an algorithm based on the knapsack formulation that maximizes the utilization and profit of the Grid infrastructure. This means, that not all replicas initially defined would be scheduled in case that no resources are available. The paper presents results obtained from simulation. The work in [54] proposes replication as a mean to obtain high availability in Grid taking into account economy and market aspects.

In the public-resource computing system BOINC [3], a redundant computing mechanism is implemented to deal with erroneous computational results. This erroneous results may arise from malfunctioning of computers or occasionally from malicious participants. In this case, the system specifies N redundant instances for each work unit. When $M \leq N$ of these instances have been executed, an application specific function is called to compare the results and possibly select a canonical one. If no consensus is found, new instances will be generated for the work unit, and the process continues until a consensus is found or a maximum timeout limit is reached.

Regarding workflows, in [113] the authors present and extensive review of the existing workflow systems, including the fault tolerance techniques applied on them. In [50] a flexible failure handling framework is presented. It is composed of two steps, a task failure detection and the recovery methodology. For the recovery phase, integrated task-level and workflow-level failure recovery techniques are applied. The task-level techniques (retry, replication and checkpointing) are intended to mask task crash failures in the task level, i.e., without affecting the flow of the encompassing workflow structure. The workflow-level techniques are intended to apply in the workflow level (i.e., by allowing changes to the flow of workflow execution) to handle the user-defined exceptions as well as the task crash failures that cannot be masked with the task-level techniques. Its main drawback is that a user must implement the fault tolerance strategy by analyzing each case in the application. ASKALON [32] and P-GRADE [53] are other cases of workflow systems that integrate fault tolerant features.

2.4.3 Comparison with GRIDSs mechanisms

GRID superscalar is a system which runs workflow applications. These applications are formed by many tasks which are parts of the whole application. A Grid level fault tolerance is able to deal with failures in a single task, but it does not have the global point of view of the application to implement advanced policies to tolerate failures for a particular application. Therefore, in our case, Grid level mechanisms can help, but are not enough to achieve our objectives.

In the application level two types of errors can be treated: application errors and Grid errors. Our work focuses only in Grid errors, as application errors in our system can be treated when programming the application (which will generate an error-aware workflow). If we deal with Grid errors with the global point of view of the workflow application, we can use a more optimal strategy to achieve fault tolerance for the application following our objectives:

- Be able to save the work completed by the application before an unrecoverable failure happens.
- Keep running an application when permanent or temporal failures arise in the Grid.
- Implement reactive (retries) and proactive (replication) strategies based on the application workflow.
- Take advantage of replication mechanisms to achieve speed up applications also.
- Make the mechanisms almost transparent for end users.
- Cause minimum overhead to the system.

We have seen related work dedicated to achieve fault tolerance for workflow applications. However, the main difference with our approach is that they propose that the user decides in advance which fault tolerance techniques should be applied each time (alternative machines to retry a task, ...), while in our case, the runtime system will automatically decide in each case what to do at run time. Besides, we exploit the capability of replicating tasks not only to ensure fault tolerance, but also to increase the performance of the applications run in an heterogeneous Grid. To our knowledge, this is the first time replication is applied to these two goals.

Chapter 3

GRID superscalar: easy programming model for the Grid

GRID superscalar (shortened GRIDSs), is a new programming paradigm for GRID enabling applications, composed of a programming interface and a runtime. The process of generating and executing an application with GRID superscalar is composed of a static phase and a dynamic phase, corresponding to the programming interface and the runtime respectively. In the static phase, the application is programmed using the GRIDSs API, the operations to be executed in the Grid are selected, and the application's structure is converted to a master-worker paradigm. In the dynamic phase the GRID superscalar runtime library and basic Grid middleware services are used. The application is started in the localhost the same way it would have been started originally. While the functional behavior of the application will be the same, the basic difference is the fact that the GRID superscalar library will exploit the inherent parallelism of the application at the coarse grain task level and execute these tasks independently in remote hosts of the computational Grid.

This chapter describes in detail the GRID superscalar user's interface. The design of the programming model and its objectives are presented in the first section, as well as the target applications that we want to tackle, the architecture of a GRID superscalar application, and the concept of using our programming model in different programming languages, as it is done in assembler languages.

The second section describes the programming model interface, to allow to understand how an application must be programmed in our system. It is presented how to describe with the IDL file the tasks to be executed in the Grid, what adaptations in the code are needed, and how the API must be used. Some advanced features such as the speculative execution of tasks and the possibility of defining the constraints and cost of a task are described from the user's interface point of view. Finally, a summary of the programming considerations and tuning capabilities of the application are presented, together with a summary of the steps needed to program an application with GRID superscalar.

The third section compares our programming model with other programming environments to help to understand the simplicity achieved. We first compare how a user should submit tasks with Globus in order to see the difficulties of using a Grid middleware without a programming model. Then, we compare our interface with DAGMan, Ninf-G and the VDL, which have been introduced in Chapter 2.

3.1 Design

Grid middlewares offer a powerful set of services in order to exploit the capabilities of the Grid, but they are developed at a very low level, which is far from what users expect as a tool to develop their applications. Moreover, there are many different Grid middleware solutions which may be used, such as the Globus Toolkit [33], Unicore [31] or gLite [60], and if users select one of them to be used to program their application, the portability of this application to a new Grid middleware would mean to develop the application again from scratch in the new environment. An extra layer is needed in order to allow users to develop their applications in a generic way, not for a particular Grid middleware.

With the objective of covering this gap, many research groups have worked in creating programming models for the Grid. While the objective of shielding the user about knowing the details of how a Grid middleware must be used is common in all of them, they include other objectives which make them different, as it has been discussed in Chapter 2 with some existing programming models. Programming models are composed by a programming interface and a runtime which implements the functionality offered by the model. In this chapter, we are going to focus in the objectives of the programming interface.

By analyzing the previous work related to programming models for the Grid, in our case, we have identified four main objectives:

- Maximum simplicity when programming.
- Exploit the parallelism of the programmed application.
- Keep the Grid invisible to the user.
- Offer advanced functionality.

The objectives of having a very simple programming interface and exploiting the parallelism of the application seem to be contradictory, as a parallel programming paradigm should be used in order to program a parallel application, which is not a simple paradigm. However, we will see that we are able to provide a sequential programming paradigm without any specific calls to enable the parallelism or manage the work to be done, which is able to exploit the implicit parallelism of the application. Existing programming models and tools include features to exploit the parallelism of the application but this parallelism is always explicitly described by the user when programming the application: by drawing a workflow in a graphical tool which describes the tasks to be performed by the application, by explicit calls to open or close parallel regions in the code, or by managing entities which can run in parallel. There are only two references in the literature which deal with exploiting the parallelism of an application implicitly from a sequential code. The first one is ProActive [13], but it is limited to an internal object that can only be run in a single machine (thus, parallelism with threads in a single machine), and the second one is Swift [114] which did not exist at the beginning of this thesis, as it has been proposed recently.

Another important aspect is the size of the API defined. Simplicity means that the API we offer to users must be minimal, or even nonexistent. Having a very small API also has an influence in the learning process of a programming model: the less the calls defined, the faster the users will learn them. We can find in the literature very powerful programming models such as ProActive but with so many defined calls to use them, which increases their difficulty. Our API must be so small and simple that the process of adapting an existing application to our model could be done almost automatically.

One of the key features we want to provide to users is the capability of keeping the Grid as invisible as possible to the end user. Whenever the application is programmed, nothing about where a particular process must be run, or where some data must be moved to operate should be specified. Our programming model must be *Grid unaware*. The vast majority of programming tools for the Grid follow this approach, but we find some tools such as Triana [99] where users must specify explicitly data movement or in which machine a particular part of the application must be run. The decision of creating a Grid unaware programming model affects not only to the user interface, but also to the runtime design, as the runtime will be in charge of deciding data movement and job distribution among machines.

Our simplicity concept must not exclude the ability of offering powerful functionality to users. This means that any functionality we define will be implemented transparently to end users, or will require the minimum interaction with them.

As no other programming model follows all our objectives, we have created our own programming model from scratch called *GRID superscalar*.

3.1.1 Target applications

Following the goals already presented in this chapter, our programming model tries to tackle applications that are able to increase their performance when executed in a Grid environment. The way to increase this performance is by means of executing parts of the application in parallel over the different resources the Grid provides. This means that the algorithm implemented must be suitable to be splitted in parts or stages that have enough computing demands to consider the Grid as a good platform to solve them (always considering the overhead the Grid may introduce in the execution because of its distributed nature). Thus, these tasks must be *coarse grained* tasks. They also need to be independent tasks which operate on the parameters and local data, without any concurrent communication between them while they run. Nevertheless, they can get the results of other tasks as inputs. Common examples of this type of applications are scientific workflows, optimization algorithms and parameter sweep applications.

A scientific workflow is a new special type of workflow that often underlies many large-scale complex e-science applications such as climate modeling, structural biology and chemistry, medical surgery or disaster recovery simulation. Compared with business workflows, a scientific workflow has special features such as computation, data or transaction intensity, less human interaction, and a large number of activities. These activities are connected forming what is called the *workflow*, which describes how the execution flow must happen when the whole application is executed. Another common name for the activities described is *task*. Thus, we can say that workflows are applications

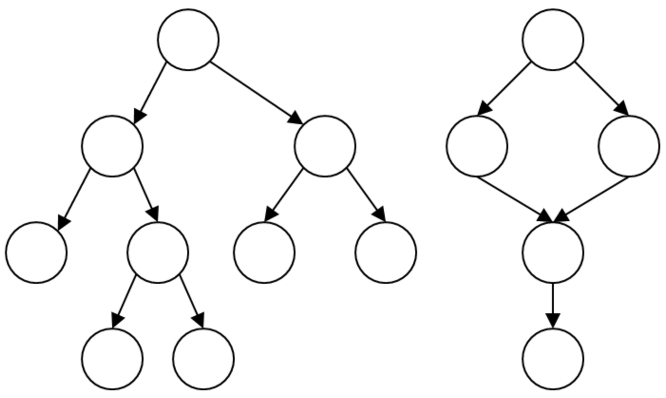


Figure 3.1 A workflow structure example

composed of tasks with a certain order between them. An example of a workflow is shown in Figure 3.1, which contains several tasks and arrows which describe the execution flow.

The common way to describe a workflow is a directed graph, where nodes represent the tasks to be performed, and edges represent dependencies between those tasks. Having directed edges is mandatory because the graph is describing an execution order and to be able to identify it, we must know which task must be first in a connected couple. However, the graph may be cyclic or not, considering what the dependencies mean in the workflow. Dependencies are commonly divided in:

- Control flow dependencies
- Data flow dependencies

Control flow dependencies are those exclusively referring to an arbitrary order between tasks. Conditional edges and loop edges are the main control flow dependencies. More specifically, loop edges determine that the workflow is a cyclic graph. When control flow dependencies are used, a task corresponds to the definition of an operation and it can be invoked several times (it is a generic representation of what must be executed). Data flow dependencies strictly describe how data is required among tasks. Therefore, an edge will exist only if a task requires data that another task generates, so it cannot start its execution until the needed data is generated. In a graph with only data flow dependencies, a task represents a single invocation of an operation (each task is only executed a single time). Thus no cycles can be formed, as a new invocation of an operation would mean a new task in the workflow. The graph is a directed acyclic graph (or DAG).

Optimization algorithms are very used in a wide spectrum of scientific fields in order to find optimal solutions for problems which can vary from finding the optimal network structure in a chemical reaction system, to obtain the optimal configuration of a network in order to reduce the execution time of an MPI application. The common structure of these algorithms is to simulate a solution several times changing some parameters, and after those simulations are completed, an evaluation step is needed in order to determine which one of them is nearest to the optimum. Then, the best solution is taken as a reference to generate new parameters that will form new simulations to be executed and evaluated. This will be done until the solution reaches the optimal criteria established. If the computing demand of the simulation is big, we can easily see that these type of applications are good candidates to be executed in the Grid, as they can also be splitted in tasks (a task may be one or several simulations). They are also good candidates to be parallelized, since between evaluation steps all tasks can be executed in parallel.

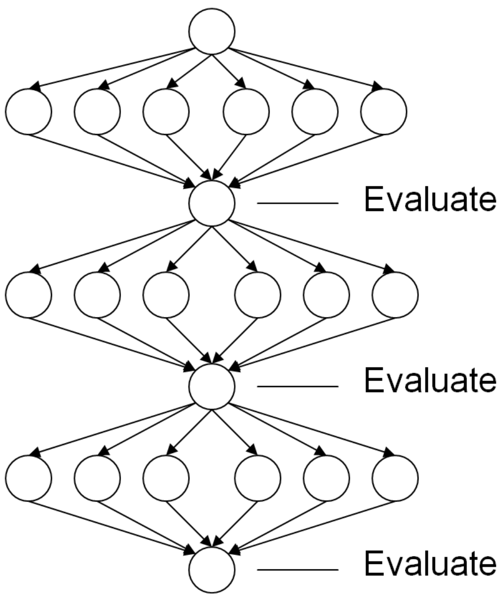


Figure 3.2 Optimization workflow example

Figure 3.2 shows the workflow of a possible optimization algorithm, where we can find parallel regions and a synchronization point in every evaluation step.

Parameter sweep applications are very similar to optimization algorithms but with some differences. The main difference is that simulations are not directed with evaluation steps. The algorithm structure is defined with a loop where first new parameters for a simulation are defined, and then the simulation is executed using the new parameters. If the simulation's result does not reach a certain criteria, a new simulation must be submitted. As with optimization algorithms, parameter sweep studies are suitable to be executed in the Grid as they can be splitted easily in tasks, and those tasks are suitable to be executed in parallel, because no data dependencies exist between different simulations. In this case the parallelism would be massive, as the whole application does not have a single dependence between its tasks. We can see that in Figure 3.3: each evaluation depends on a single task, which is independent from the rest.

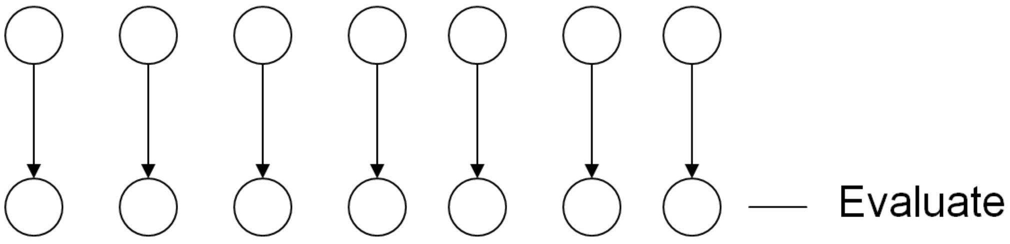


Figure 3.3 Parameter sweep workflow example

In GRID superscalar, workflow applications can be implemented using a sequential language, and the sequential applications previously described can be transformed automatically in workflows, taking care about their internal data dependencies. We may also notice that the majority of applications that can be easily splitted in tasks and have a suitable computation demand for the Grid are those which call external simulators, finite element solvers, bioinformatic applications such as BLAST [18] or chemistry applications using the GAMESS [38] package. In all such cases, the main parameters of these tasks are passed through files. This is our rationale to consider files as the main data to be taken into account for the automatic workflow generation that will take place in the runtime of our programming model, and will determine the design of the interface. We will see in Chapter 4 that this decision will also ease the implementation of the dependence detection, although extending the detection to parameters which are not files could be considered in the future.

Another important fact is that we want our programming model to follow a sequential way of programming, albeit its execution must be parallel. The parallel execution of complex applications has proved to be an excellent way of reducing the time to get results. However, the scientific community has invested lots of efforts in porting or writing the applications in parallel programming models such as MPI [66]. MPI's programming interface is not very easy to use for non-expert programmers, as the parallel execution must be handled explicitly when programming the application[1]. GRIDSs follows a sequential programming paradigm in order to ease the way it must be programmed, in contrast to parallel programming paradigms. The simplicity of this sequential paradigm also contributes to a reduction of the time needed for a user to learn how to use the programming model.

An easy way to understand our goal is to consider a sequential algorithm, that includes several calls to functions. In GRIDSs, a function will correspond to a task and will be executed in the Grid concurrently with other functions, achieving the parallel execution of the application transparently from the programming paradigm point of view. We may say that GRIDSs achieves the *functional parallelism* of the application.

An example of an application written with GRID superscalar is presented in Figure 3.4 and Figure 3.5. The first figure presents the master side of the application, where we can see some calls to the API provided in the programming model (described in Section 3.2.3). The second figure corresponds to the worker side of the application, which is the function that will be executed in the Grid. These are some examples of applications that our programming model targets, but that is not only limited to. In general, we may say that

[1]We provide MPI as an example of a well-known parallel programming paradigm. However, we are not comparing GRIDSs and MPI, as their objectives are different and they target different applications

```
#include "GS_master.h"
#include "matmul.h"

int main(int argc, char **argv)
{
  char f1[15], f2[15], f3[15];

  GS_On();
  for(int i = 0; i < MSIZE; i++)
    for(int j = 0; j < MSIZE; j++)
      for(int k = 0; k < MSIZE; k++)
      {
        f1 = "A." + i + "." + k;
        f2 = "B." + k + "." + j;
        f3 = "C." + i + "." + j;
        //f3 = f3 + (f1 * f2)
        matmul(f1, f2, f3);
      }
  GS_Off(0);
}
```

Figure 3.4 GRIDSs application example (master)

```
void matmul(char *f1, char *f2, char *f3)
{
        block<double> *A;
        block<double> *B;
        block<double> *C;

        A = GetBlock(f1, BSIZE, BSIZE);
        B = GetBlock(f2, BSIZE, BSIZE);
        C = GetBlock(f3, BSIZE, BSIZE);

        A->mul(A, B, C);

        PutBlock(C, f3); //A and B are sources

        delete A;
        delete B;
        delete C;
}
```

Figure 3.5 GRIDSs application example (worker)

we want to cover any application or algorithm where the work to be done can be splitted in tasks with a sequential programming fashion, such as branch and bound computations, divide and conquer algorithms, recursive algorithms, and so on.

In general, with GRIDSs a sequential application composed of coarse grained tasks is automatically converted into a parallel application where the tasks are executed efficiently in different machines of a computational Grid.

3.1.2 Application's architecture: a master-worker paradigm

In this section we are going to see how a sequential application is automatically converted in a parallel application when using GRIDSs. We have previously presented that our objective is to tackle applications that can be splitted in parts suitable to be distributed in the Grid, and this is what determines that the best parallel execution paradigm for our programming model is a master-worker paradigm.

The master-worker approach has its basis in the division of the application in two main parts: the master and the workers. A master part of the application is in charge of splitting the work to do in parts in order to distribute them to the workers. Thus, the master part does not perform any computation work at all, that is let to be distributed to the workers. The master is in charge of coordinating all the execution: generate tasks splitting the work to do in parts, send the work to the workers and gather results when workers finish their execution.

In our programming model, the master-worker paradigm fits perfectly with our objectives: the master part of the application will be formed by the main program of the application. This main program will describe the algorithm that must be executed, and will include calls to functions that will be executed in workers distributed in the Grid. GRID superscalar applications will be composed of a master binary, run on a host, and one worker binary for each worker host available in the computational Grid. However, this structure will be hidden to the application programmer.

To transform a sequential application into a master-worker application, we need to define the glue that will allow us to do so. In order to achieve the remote execution of the functions, we follow a Remote Procedure Call approach, where in the main code, instead of calling to the original function, the call is done to a new function (a wrapper) that will start all the intermediate steps in order to call to the original function in a remote machine. This is done by using function stubs in the master side of the application, and skeletons in the worker side.

The master stubs consist of a file with wrapper functions for each of the functions that must be executed in the Grid. When the master program is compiled, this file will be used instead of the original tasks' file and, therefore, these wrappers will be the ones executed instead of the original user function whenever the main program calls them. The wrappers are in charge of redirecting the function call to the GRIDSs runtime. This means that the parameters of the function are used to call to the *Execute* function, which is the main primitive in the runtime called from the user interface that generates a new task inside the runtime (a new function call). Section 3.1.3 will describe the usage and behavior of the Execute call more in detail.

Figure 3.6 presents an example of a stub file generated when the C language is used to program the application. We will see again in section 3.1.3 that several programming languages can be used thanks to the Execute call.

```
#include <gs_base64.h>
#include <GS_master.h>
#include "matmul.h"
int gs_result;

void matmul(file f1, file f2, file f3)
{
   /* Marshalling/Demarshalling buffers */
   /* Allocate buffers */
   /* Parameter marshalling */
   Execute(matmulOp, 3, 0, 1, 0, f1, f2, f3, f3);
   /* Deallocate buffers */
}
```

Figure 3.6 Sample stub file

The worker skeleton is the main program of the code executed in the workers. With the skeleton, a binary is generated that will be called from GRID superscalar's runtime by means of a Grid middleware (details about all this process will be described in Chapter 4). The worker code is basically a conditional structure which depending on a parameter received on invocation will call the corresponding original user function. Before calling the user functions, the parameters passed to the skeleton are retrieved to be passed to them.

Figure 3.7 presents an example of a skeleton file generated when the C language is used to program the application. We can see a case structure is used to determine the operation to be executed. IniWorker and EndWorker are auxiliary functions defined in a GRIDSs worker library to make some initializations and finish the worker execution respectively.

```
#include <gs_base64.h>
#include <GS_worker.h>
#include "matmul.h"

int main(int argc, char **argv)
{
  enum operationCode opCod = (enum
      operationCode) atoi(argv[2]);
  IniWorker(argc, argv);
  switch(opCod)
  {
  case matmulOp:
    {
    matmul(argv[3], argv[4], argv[6]);
    }
    break;
  }
  EndWorker(gs_result, argc, argv);
  return 0;
}
```

Figure 3.7 Sample skeleton file

In order to execute a GRIDSs application, a master-worker application must be previously built and distributed in the machines where it must be run. This requires that several binaries are generated before the application can be started. More precisely, two types of binaries must be generated, one for each role a machine may have during the execution of the application: a binary for the master machine, and a binary for every worker machine. The master binary is the process where the whole execution must start and it must be placed in a single machine. When the master binary is executed, the algorithm execution starts, and a worker binary will be invoked by means of the Grid middleware for every task generated.

Figure 3.8 shows how files are linked when using C to obtain the final application binaries. In the master, the original main program (app.c) is linked with the stubs (app-stubs.c) and with the GRID superscalar runtime library in order to generate the master binary executable. In the workers, the skeleton (app-worker.c), the file with the code of the original user functions (app-functions.c) and the GRID superscalar worker library are linked to obtain each worker's binary. The "app.h" file is a header file which contains the C definitions for the functions that must be executed in the Grid. These definitions are needed in both the master and the worker binaries generation.

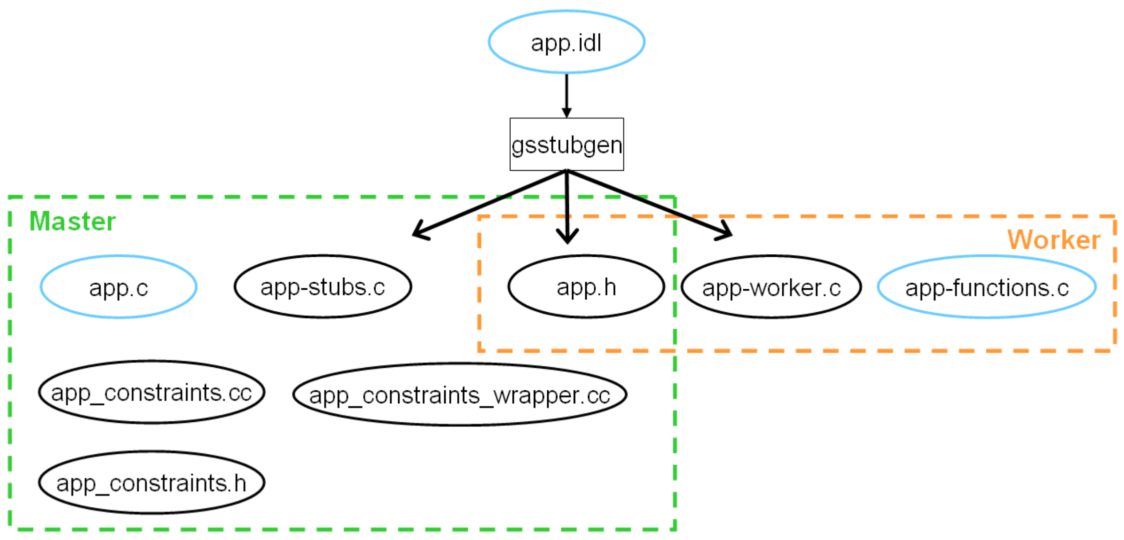


Figure 3.8 File linkage to create the master and the worker

The other three files seen in the figure are used for function cost and constraints specification, described in section 3.2.5. The "app_constraints.h" contains definitions used by the other two files. The "app_constraints_wrapper.cc" contains glue code that is needed by the GRID superscalar runtime library. Finally, the "app_constraints.cc" contains functions that determine the cost and constraints of each function to be executed in the Grid.

Initially the generation of the stubs, the skeletons, as well as the compilation and distribution of the master and worker binaries was done by hand. However, work done mainly by other members in our research group allows to automate these steps by means of a set of tools known as *gsstubgen*, *gsbuilder* and *deployment center*. The gsstubgen tool generates stubs and skeletons for an application. The deployment center is a graphical tool that mainly allows to copy to remote machines the source files needed in order to build the master or worker binaries using the gsbuilder. It also allows to create the *Grid configuration file*, where the machines and their characteristics are specified. More details can be found at [73].

Figure 3.9 shows the sequence of calls of an application run without GRID superscalar. The application is run sequentially on a local host. In Figure 3.10 the sequence of calls of an application run with GRID superscalar is shown. In this case, the main program together with the corresponding stubs file is executed in the localhost. The worker binary built with the corresponding skeleton file and the original user functions, is run in the remote hosts. When the main program calls one of the functions that must be executed in the Grid, the wrapper function in the "app-stub.c" file will be executed instead of the

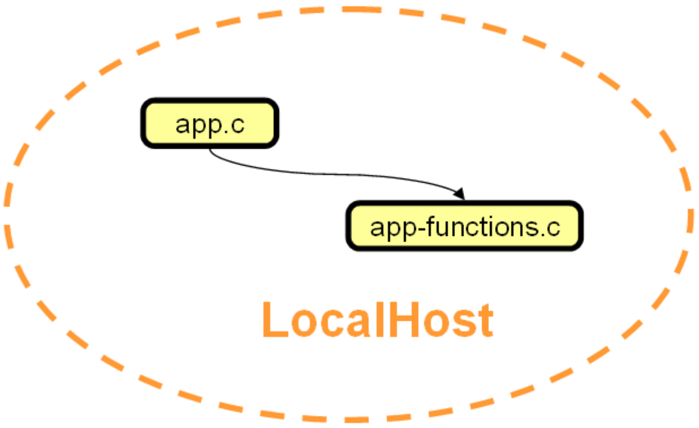


Figure 3.9 Sequence of calls without GRID superscalar

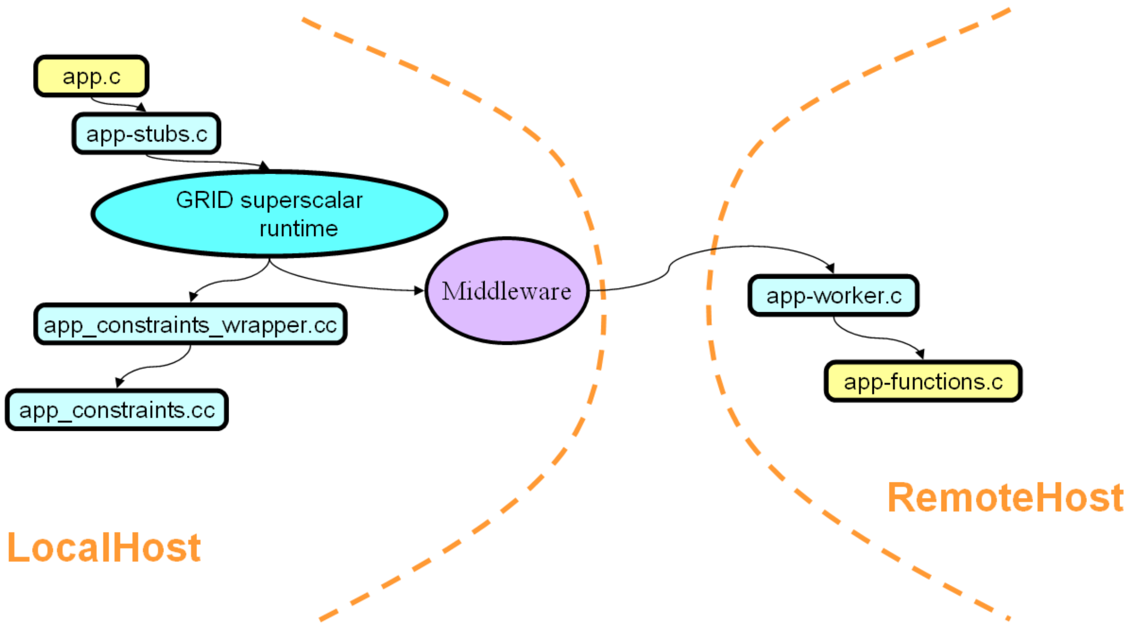


Figure 3.10 Sequence of calls with GRID superscalar

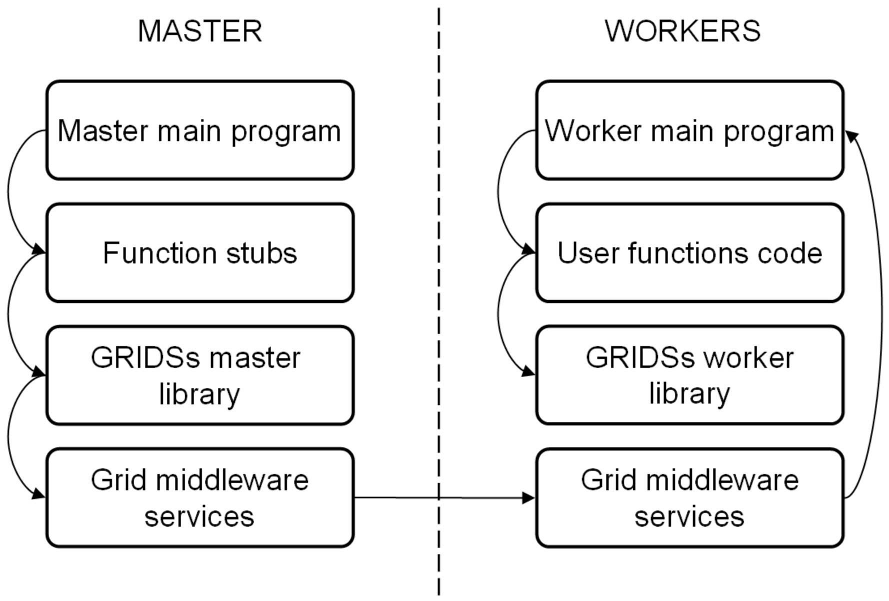


Figure 3.11 GRIDSs application's architecture

original implementation. This wrapper function will call the GRID superscalar "Execute" function, that will generate a task. We will see in Chapter 4 that whenever the task is ready for execution (it has no data dependencies), an extra step is needed to check the constraints and evaluate the cost of the task. The user provided functions in "app_constraints.cc" file are used for that purpose.

Finally, the task is submitted for execution in a selected resource using the Grid middleware. The worker program is run in a remote host. From this worker program, the original user task will be called and executed. When the task finishes, the runtime is notified by the Grid middleware. This will allow the GRID superscalar runtime to update its data structures and continue the application execution, as we will see in Chapter 4.

As a summary, the GRID superscalar application's architecture picture is shown in Figure 3.11, specifying the different layers needed to generate the resulting master-worker application. Arrows show the time order in which the different layers are called. We must remark that this new built master-worker application has the same functional behavior of the initial user application.

3.1.3 The Execute generic interface

We have mentioned in previous sections that the main interface the runtime offers as an entry point is the *Execute* call. We have seen too how the Execute call is invoked from user's main code in the master through function stubs, and how a user function is run in a worker thanks to the skeletons. In this section we are going to present the main benefits provided by having a single call for task generation from the point of view of the user's interface.

One of the concepts that GRID superscalar pursues is the *intermediate language* concept. This concept is seen in the assembler code: programmers use their preferred programming language to implement their application, and thanks to a compiler, the code is translated into assembler code that can be translated into machine code in order to be run. The same way, GRID superscalar pretends to translate an application implemented with a particular programming language to an intermediate representation that can be later executed in the Grid. The intermediate representation in GRIDSs is the data dependency graph created at run time (see Chapter 4), which would correspond to the assembler code. However, while in assembler the instruction set (the operations that can be used) is clearly defined in advance, in our programming model the instruction set is different for every application. This means that we need a generic interface able to specify what function must be executed when used, and what parameters are needed. To solve this, the Execute call is defined.

The Execute call has been defined as a generic call, where an identifier of the function to be executed is passed, together with four numbers that specify how many parameters of each type are passed, and the parameters themselves (a variable list). Parameters passed to Execute are converted to strings, so the call receives only integers and strings as parameters. Therefore, in the instruction set of our particular *assembler code*, Execute is the one and only operation defined, but generic enough to contain the future definitions of operations made by end users when programming. In section 3.2.2 we will see how these operations can be defined by users. Figure 3.12 shows the definition of the Execute call.

```
void Execute(int OpId, int nSour, int nSourGen,
     int nDest, int nDestGen, ...)
```

Figure 3.12 The Execute call

Although the GRIDSs runtime is implemented in C++, the Execute interface, together with the stubs, allow an easy building of different language bindings to program an application in our system. In order to create a new language binding, the main requirement is the possibility to call to the C Execute routine from the new language when the master stubs file is generated, as this file will be written in the new language. In the worker side, as the skeleton is a main program itself which generates a binary, things are simpler. The skeleton will be implemented in the new language, as well as the user functions, and the invocation will be done as if the worker is an independent program. This has been demonstrated by other members in our research group by building bindings for Java, Perl and Shell Script languages for GRIDSs [52]. Another approach that could have been implemented would be to develop a compiler for GRIDSs applications and a single programming language. A source to source compiler could translate the programmer's code in a code that the runtime understands. However, basing the code generation in a compiler instead of having stub files as intermediates limits the possibilities of using several programming languages in the system, because whenever a new language wants to be supported, a new compiler must be created for it, which is not a trivial task.

The possibility of programming applications with GRID superscalar using different programming languages is a powerful fact in terms of technology adoption. If users are able to program using our system with their preferred programming language, the learning process in order to use GRIDSs will be reduced. We have seen in Chapter 2 that many existing programming models only focus on a single programming language, or they even define a new procedural language to be used in order to implement an application. Nevertheless, many scientific fields have legacy code written in a particular language that is commonly used in the field. While in other systems porting those legacy codes would mean to re-implement the application or develop wrappers that execute the legacy binaries, in GRID superscalar a new language binding can be easily developed, so the option of directly using the legacy code is available.

3.2 User interface

In order to define the user interface needed to program applications with GRID superscalar, we must keep in mind the objectives we have previously presented in Section 3.1. The interface must follow a sequential programming paradigm and no parallelism is needed to be expressed in the algorithm, to make it simple. The Grid must remain as invisible as possible for the end user, specially for the application's code. Finally, any advanced functionality that we want to provide to users must be integrated in the programming model in the simplest way, or even it must be transparent to the end user. These objectives make easier not only the way to program the Grid, but also to port sequential applications to be executed in the Grid, as minimum changes will be needed to adapt the application to our programming model. Next subsections show how an application must be programmed with GRIDSs. This demonstrates why we can state our programming model is very simple to use, and it will allow us to develop a section comparing how applications are programmed in other Grid programming models (see section 3.3).

To develop an application in the GRID superscalar paradigm or to adapt an existing application to it, a programmer must go through the following three stages:

- Task definition: identify those functions/programs in the application that are going to be executed in the computational Grid.
- Task parameters definition: for every task defined, identify which parameters are input/output files and which are input/output scalars.
- Write the sequential program (main program and task code).

The task definition and task parameters definition are performed by writing an interface definition file (IDL file) with the selected functions to be run in the Grid (sections 3.2.1 and 3.2.2). The API provided to write the application is presented in 3.2.3).

3.2.1 Task definition

In application programming, there are some options available when structuring the code. One really useful way is to program functions, instead of programming everything in a big main function. This helps in two ways: it makes the code easier to understand, and allows to repeat the same functionality in other stages of the application. In GRID superscalar the organization of the main program in functions is mandatory for those parts of the code that must be executed in the Grid, because conceptually the body of the existing functions in the application is what can be distributed. Those parts can be already in a function, called from the main program. But if this is not the case, that part of the code must be defined in a local function, even if that function is only going to be called once.

To determine what parts of the application should be run on the Grid there are two main scenarios. The first scenario consists of a program that cannot be fully executed on the local machine because it requires more resources than there are locally available (i.e. high memory demand). In those cases the target functions are those functions that cannot be executed locally. The second scenario is composed of those cases in which more performance is desired than there is locally available. In this scenario the target functions are those that consume most CPU time and are executed most often. Both scenarios show a need of externalizing the computing demands of the application to more powerful machines. In some applications, profiling or tracing tools may be used to assist the user in the selection of the functions [59].

There is an important requirement when defining the header of the function. In the header, all parameters used must be specified: the files needed (files read and/or written inside the function) and scalar parameters needed (read and/or written) (i.e. a scalar value could be needed as a parameter to start a simulation). Return values of the function (files or scalars), must be also specified in the header, as results of tasks will be asynchronously returned when tasks finish their execution in the Grid, thus not allowing functions to have a direct return value. The data types which can be used as parameters are defined in section 3.2.2.

In order to understand better the steps needed to program an application with our system, we present a matrix multiplication example. One common operation done between matrices is the multiplication. When matrices grow in size, it also grows the execution time of the multiplication, thus a way to speed up this computation is needed (parallelizing the code). A first step is to divide matrices in blocks, forming what is known as a hyper-matrix (a matrix made of matrices). This provides several advantages compared to a version without block division, as a full row or column is not needed to do some calculation, because blocks can be operated between themselves. Another advantage is that it is not needed to have all matrices in memory, because for a single calculation only the blocks that are going to be operated are needed. Blocks are stored in the disk, and when a calculation is going to be performed, the blocks required are read again into memory, just to be stored again when the calculation has finished. This technique is known as an out-of-core implementation. The source code of the matrix multiplication has three local functions: PutBlock, GetBlock and matmul. The function that we want to execute in the

```
void matmul(char *f1, char *f2, char *f3)
{
        block<double> *A;
        block<double> *B;
        block<double> *C;

        A = GetBlock(f1, BSIZE, BSIZE);
        B = GetBlock(f2, BSIZE, BSIZE);
        C = GetBlock(f3, BSIZE, BSIZE);

        A->mul(A, B, C);

        PutBlock(C, f3); //A and B are sources

        delete A;
        delete B;
        delete C;
}
```

Figure 3.13 The matrix multiplication function

Grid is matmul (Figure 3.13), and the other two are auxiliary functions needed by matmul to save or restore a block from disk.

In the code, the BSIZE parameter determines a dimension of the block in elements. By analyzing the parameters needed inside the function we can determine that, in this case, the definition of the function is already correct. This is due to having the three blocks read (f1, f2, f3) and the one written (f3) already specified in the header. It is specially important in this case to have the three blocks specified, as we have seen in previous sections that the GRID superscalar main data type is the file. In a local environment, the file could be omitted in the header, and accessed directly in the function, but this does not work in GRIDSs, as the function will be executed in a remote environment, where the file may not be initially available.

3.2.2 Interface Definition Language (IDL)

The GRID superscalar programming model must know somehow which functions in the code have been selected. The Interface Definition Language (IDL) file is a means of communication between the developer and the GRID superscalar library. Since the GRID superscalar runtime detects at run time the data dependencies between the tasks that must be run in a Grid (see Chapter 4), the information about which are those tasks and the direction and type of their parameters must be known in advance. This information is provided in the IDL file. The syntax used to describe the tasks and their parameters is a simplified interface definition language based on the CORBA IDL standard [51]. We have selected this language because it offers an elegant and easy way of describing the tasks' interface including the direction of the parameters. There is not any other relation between GRID superscalar and CORBA (no tool from CORBA is used).

Each function that the user identifies as a candidate to be run in the Grid must be described in the IDL file. The rest of the functions in the code are not specified here, so

```
interface MYAPPL {
  void myfunction1(in File file1, in scalar_type
     scalar1, out File file2);
  void myfunction2(in File file1, in File file2, out
      scalar_type scalar1);
  void myfunction3(inout scalar_type scalar1, inout
     File file1);
};
```

Figure 3.14 IDL generic syntax

they can be executed locally in the master or used as auxiliary functions in the worker. The list of parameters of the function is also specified, indicating their type and if the parameter is read inside the function (in), written (out) or both read and written (inout). The types supported are divided in two main categories: files and scalars. Figure 3.14 shows the generic syntax of an IDL file.

Files are a special type of parameters in the IDL file. Our programming model focuses in files as the main data parameter for the Grid tasks, and files define the tasks' data dependencies. A file is also needed to distinguish between a regular string passed as a parameter to the function, or a string passed which represents a file. For those reasons, a special type *File* has been defined. Several types of scalars are supported, such as char, wchar, string, wstring, short, int, long, float, double and boolean.

The definition of a function in the IDL file must be exactly with the same parameters as the real function it represents. The main difference is that in the IDL the files are tagged with the special type File, and the direction of the parameters is also included. As we have mentioned in previous section, no return value can be specified in the function. This is confirmed in the IDL file by only accepting *void* as returning types of the functions.

Following with the matrix multiplication example used in previous section, we can see how the IDL file should be defined in that case. Figure 3.15 shows the solution. In this example there are two input files, and an input/output file (where the multiplication is going to be stored).

```
interface MATMUL {
  void matmul(in File f1, in File f2, inout File f3);
};
```

Figure 3.15 matmul IDL

3.2.3 Programming API: master and worker

The most important feature when defining the API of our programming model is that it must be very simple to use: that is why a very small set of calls is defined. One of the main goals is that the Grid must remain invisible to the user when programming the application, therefore the API cannot include any references to where a task must be executed or how data needed in that task is handled for its correct execution. As our programming model is sequential, no parallel sections or parallel handling calls to open or close parallelism will be defined. We have seen previously that the application is split into a master part (with the main algorithm), and a worker part (where functions to be executed in the Grid are implemented). These two parts are programmed in different files, as only the worker part will be distributed in the Grid, and each of them has some API calls defined.

The master

The main program that the user writes for a GRID superscalar application is basically identical to the one that would be written for a sequential version of the application, The differences would be that at some points of the code, some primitives of the GRID superscalar must be called, mainly to start and stop the runtime, and file handling primitives on those parts where files are read or written.

If we think about the main program as a scientific workflow, we may see that the workflow can be easily described with a sequential program. Any kind of repetitive behavior (loops) or conditions can be expressed with the corresponding imperative language (C/C++, Java, ...) allowing a high degree of flexibility. Moreover, users can include inline code or calls to auxiliary functions, and not only the defined tasks for the workflow (the IDL functions).

The basic defined calls are:

- GS_On(): notifies the GRID superscalar runtime that the program is beginning, so the runtime is initialized. The best place to put this call is at the beginning of the main code, but it can be placed later, always considering that no GRID superscalar primitive or IDL function can be called before GS_On.

- GS_Off(code): this call will wait for all remote tasks to end, and will notify the GRID superscalar runtime to stop. It is usually placed at the end of the code, so no more calls to the runtime or IDL functions can be done after it, but other local code can be executed. The passed argument is used to notify error situations. Passing a 0 parameter indicates that no error has been detected. In order to indicate an error a code -1 must be provided. This will safely stop the master program.

- GS_Open(filename, mode) and GS_FOpen(filename, mode): GRID superscalar needs to have full control over the files, as they are the main data considered. These primitives allow to work with files while GRID superscalar can control data dependencies. They both return a descriptor that must be used in order to work with these files (i.e. to call read/write functions). These descriptors correspond to the ones returned by the C standard library (when using open and fopen), so there is no need to change following C library calls that work with these file descriptors. Modes supported are: R (reading), W (writing) and A (append).

- GS_Close(file_des) and GS_FClose(file_des): these primitives must be called to close the files opened with previous calls. The previous file descriptor returned by GS_Open or GS_FOpen primitives must be used here as a parameter.

The implemented functionality of those functions is required for a correct behavior of the GRID superscalar applications. This set of primitives is very simple, and easy to understand for end users, so using them in the master code is straightforward. Adding these calls to the main program could be even automated, by substituting the C standard library symbols of the calls to open and close in the program, or creating a source to source compiler that adds and substitutes these calls. This is proposed as future work of this thesis.

The main extra functionality of the open and close functions defined is to notify the runtime that an operation is going to be performed in a file. GRIDSs must be aware of all operations done to a file, as it is the main data used to keep the data dependencies between all tasks. Open calls can define synchronization points, because if a file to be read is generated in a remote machine as a task result, the execution will wait until that file is generated and copied back to the master, so the open call can continue. The returned descriptor must be used to perform operations in the file. This is due to a technique applied in GRIDSs runtime: the renaming of files. This and other programming considerations are described in section 3.2.6.

In the matrix multiply example, our master is shown in Figure 3.16. MSIZE represents a dimension size of the hyper-matrix. Three nested loops iterate to first generate the names of the files to be used for the multiplication, and later call to matmul to perform the operation. In this particular case, only GS_On and GS_Off must be added. The call to matmul is defined exactly with the same parameters that in the sequential version of the algorithm, therefore no change must be done.

Figure 3.17 shows a usage example of GS_FOpen and GS_FClose. The IDL file for this example describes that task *task_A1* has as input file *f1* and as output file *f2*. After the execution of this task, the main program opens the file f2 for reading. As tasks are not executed in the same moment that are called, and since the code of the application that is not in the tasks is not under the control of the runtime, the fscanf call could have been executed before the file f2 has been written, provoking an error. With the use of GS_FOpen, the runtime will synchronize the execution of the main program with the tasks run in the Grid, and wait until the file f2 is ready.

```
#include "GS_master.h"
#include "matmul.h"

int main(int argc, char **argv)
{
  char f1[15], f2[15], f3[15];

  GS_On();
  for(int i = 0; i < MSIZE; i++)
    for(int j = 0; j < MSIZE; j++)
      for(int k = 0; k < MSIZE; k++)
      {
        f1 = "A." + i + "." + k;
        f2 = "B." + k + "." + j;
        f3 = "C." + i + "." + j;
        //f3 = f3 + (f1 * f2)
        matmul(f1, f2, f3);
      }
  GS_Off(0);
}
```

Figure 3.16 Master code for matmul

```
void main()
{
  FILE *fp, char *somedata;

  task_A1("f1", "f2");
  fp = GS_FOpen("f2", R);
  fscanf(fp, "%s", &somedata);
  GS_FClose(fp);
}
```

Figure 3.17 Opening a file for reading

With the objective of providing extra functionality, some advanced primitives are defined:

- GS_Barrier(): some algorithms may need to wait for all tasks to finish in a particular point of the application. This may be useful to make a decision and keep executing the algorithm. GS_Barrier allows to do explicit task synchronization. It is similar to what GS_Off does, but it does not stop the runtime, so more GRID superscalar functions can be called later. A misuse of this call can severely slow the parallelization of the code, as it has an effect in the task generation.
- GS_Speculative_End(my_func): defined to achieve speculative execution of tasks. Basically, it waits until a special notification has risen in a worker, or until all previous tasks have ended. At the former case, a function specified by the user (my_func) will be executed. At the latter case, the function will not be called.

In most of applications the use of the GS_Barrier primitive is not required, but it is provided for completeness. The GS_Speculative_End will be described more in detail in section 3.2.4.

The workers

The worker side of the application is implemented by the user providing the code of the functions that have been selected to run on the Grid. The code of those functions does not differ from the code of the functions for a sequential application. The only current requirement is that they must be implemented in a separated file from the main program, as the worker will be distributed in the Grid. All functions specified in the IDL file must be now implemented using the same headers (same parameters and passed in the same order). The function must be considered as an isolated piece, that will be called during the execution.

The input/output parameters defined in the function header must be used to work with files inside the function (for instance, to open and close a file). This is due to the renaming technique applied in the runtime (detailed in Chapter 4), thus no explicit reference to a file can be made. However, users are allowed to create temporary files to work inside the function that can be referenced explicitly.

One primitive and one variable are defined to be used in the worker side of the application:

- GS_System(command_line): when there is the need of calling an external executable file (i.e. a simulator), GS_System must be used. The parameter passed is the command line to be executed. It has the same behavior as the *system* call in the C standard library.

- gs_result: this is a special variable defined that can be used to pass an error code to the master, so the master can stop the execution. If it is assigned a non-zero value, the master will stop the application knowing that the user has detected an error in a task.

As with the rest of basic primitives, the use of GS_System is mandatory to ensure the correct behavior of the application, because the path where the worker is executed will be different of what the user can expect. In the end of its implementation it calls to *system*, and it returns its result. The internals of this call will be explained in Chapter 4.

The gs_result default value is 0 (that means no error is detected in the task), so its use is not mandatory. If the user detects an error which requires the application to be stopped, an error code may be assigned in this variable. This code must be higher than 0, because negative values are reserved for the GRID superscalar runtime. As the value is assigned by the user, the meaning of every value is defined by the user, thus errors can be mapped to different codes freely.

In the matrix multiply example previously presented, the function implementation does not change (see Figure 3.13). The only issue is that it must be implemented in a separated file, together with the auxiliary functions needed (as PutBlock and GetBlock). So, we present in Figure 3.18 a different example to show the usage of GS_System and gs_result. In this function, a simulator named Dimemas is called as an external binary, having as input the configuration file *cfgFile* and the *traceFile*, and as an output the *outFile*. In this case, the previous call to *system* has been replaced with GS_System. Moreover, the result returned by the GS_System call is assigned to gs_result, so in case the *system* call returns a failure, the execution of the master will stop.

```
void Dimem(file cfgFile, file traceFile, file outFile)
{
  char aux[200];

  sprintf(aux, "/usr/local/cepba-tools/bin/Dimemas -t %s
      -o %s %s", traceFile, outFile, cfgFile);
  gs_result = GS_System(aux);
}
```

Figure 3.18 GS_System and gs_result example

3.2.4 Speculative execution of tasks

The initial motivation of the speculative execution of tasks comes from optimization algorithms. A common optimization algorithm is usually implemented by calling to a simulator a high number of times since an objective is reached. Each call has different input parameters in order to simulate different situations and obtain the results under that conditions. A first implementation option is to check for the result of a simulation when it ends, before launching a new one. This is valid for a sequential environment, but is not feasible in a parallel execution model (if we have to wait for a simulation results before launching a new one, no parallelism can be applied). A second option is to make a program that calls N times to the simulator (using a parallel approach), and checks results at the end. The problem of this second option is that all N simulations must be performed, even if in the first simulations the objective was already reached. Therefore, more computation than is really needed is done, and this is not efficient. A part from optimization algorithms, other programs can benefit from the speculative execution. Those are environments or algorithms that when an event is received, change their behavior to follow with the computation, or in any construction where an exceptional event may require discarding forthcoming functions (as a branch and bound program).

The GS_Speculative_End call provides a mechanism for achieving *speculative execution* of tasks. The syntax of this mechanism is really similar to the exception treatment done in languages such as C++ or Java, as well as the objective (jump in the logical sequence of the code when something happens), but its behavior is not exactly the same. The complete interface for this mechanism is composed of two primitives: *GS_Speculative_End* in the master and *GS_Throw* in the worker. The primitive GS_Throw has been implemented as a mechanism to throw exceptions in the workers, but it is a much simpler mechanism than the one currently implemented in C++ or other languages and no type is associated to the exception.

The speculative execution mechanism allows to execute simulations until an objective is reached, but exploiting parallelism, and not wasting too much resources of what are really needed. To enable the mechanism, only the special primitive GS_Speculative_End(my_func) must be called (at the master), after having called to the functions that may rise the exception (when the objective is reached). This primitive will wait until all previous tasks end, or until an exception rises from a task, taking it as the last valid task in the region. It is important to mention that there is no primitive to mark the start of the speculative region. Speculative regions are defined between the beginning of the application and the first call to GS_Speculative_End or between different calls to GS_Speculative_End.

```
#include "GS_master.h"
#include "mcarlo.h"
#define MAXITER 5000

void myfunc()
{
  printf("EXCEPTION received from a worker\n");
}

int main(int argc, char **argv)
{
  GS_On();
  for(int i = 0; i < MAXITER; i++)
  {
    generate_cfg(newCFG); //Has a GS_Open inside
    Dimem(newCFG, traceFile, DimemasOUT);
  }
  GS_Speculative_End(myfunc);
  printf("Execution ends\n");
  GS_Off(0);
}
```

Figure 3.19 Speculative execution example: master

It also exists the possibility of calling a function when an exception arises. This is done by passing a function pointer to the GS_Speculative_End primitive. This function must accomplish some requirements: it has to be a function without a return value, and without parameters. With this function a message to know that the exception has been thrown could be printed, or a global variable could be modified in the main program, so the rest of the algorithm will be aware about the situation.

In the worker side, only GS_Throw is defined. When the situation to rise the exception is accomplished, GS_Throw is called. This primitive makes the function return, so all code following GS_Throw will not be executed.

A simple optimization algorithm is presented in Figure 3.19 to clarify the usage of the speculative execution mechanism. The function *generate_cfg* is executed locally to generate a new configuration file for a simulator. The *Dimem* function receives as an input that configuration file, together with a *traceFile* and starts a simulation whose results will be written in the *DimemasOUT* file. When an exception rises from a task, all following tasks will not be executed. So, when GS_Speculative_End returns in the code, DimemasOUT will contain the output solution that reaches the desired objective, unless we have reached the maximum number of iterations MAXITER.

```
void Dimem(file cfgFile, file traceFile, file outFile)
{
  char aux[200], double result, FILE *fp;

  sprintf(aux, "/usr/local/cepba-tools/bin/Dimemas -t %s
      -o %s %s", traceFile, outFile, cfgFile);
  gs_result = GS_System(aux);
  fp = fopen(outFile, "r");
  result = read_result(fp); /* Read result value in the
     file */
  fclose(fp);
  if(result > 50 && result < 70)
    GS_Throw;
}
```

Figure 3.20 Speculative execution example: worker

In the example the Dimem function is the only one that can cause the exception. The worker code is shown in Figure 3.20. After the simulation finishes, we check if the result is between two values in order to call to GS_Throw to notify the master that the objective value has been found.

3.2.5 Task's constraints and cost specification

A Grid is usually composed of lots of different machines. They can be from clusters to single personal computers, with different software installed, different architecture, operating system, network speed, and so on. In this sense, there is a need of expressing what elements compose this heterogeneity. If we have a description of what is available in each machine, we can therefore ask for a specific feature in our Grid. An example is the execution of an external simulator in an algorithm. This simulator may be installed only in part of the machines that form the Grid, so it is interesting to define what machines have the simulator available and what tasks in the algorithm need that simulator to be executed. That way, the machines that will be able to execute a task will be first filtered to consider only those that accomplish the requirements.

In order to allow GRIDSs to perform an efficient scheduling of the tasks forming an application, or to apply other policies in the tasks (i.e. fault tolerance), metrics about the task must be provided. We provide users the option of specifying the estimated execution time of a task, considering their knowledge of the application. With the information of what time is supposed to spend a task in a resource, our runtime is able to implement a scheduling different from a First Come First Serve or a Random approach in order to execute efficiently the application in the Grid with the objective of reducing the run time.

In GRID superscalar a user can express the computation cost and execution constraints or requirements for each of the tasks. This is performed by the use of an extra source file where they can be specified. This file was already introduced in Figure 3.8 and is generated together with the stubs and skeletons of the application by the gsstubgen tool (see section 3.1.2). For every "operation_name" function defined in the IDL file, two functions are defined in this file named operation_name_constraints and operation_name_cost, and they return a default value (true for the constraints and 1 for the cost). Both functions will be evaluated dynamically by the runtime whenever a task is considered to be executed in a resource.

In order to specify a constraint for a task, an *Expression* (basically a string) must be build with the requirements of the task, and return it in the corresponding function. The syntax of the Expression must be of the format expected by the ClassAds library [77] (similar to the C/C++ syntax, with literals, aggregates, operators, attribute references and calls to built-in functions). The constraints of a task can be such as that a given software, a given operating system or architecture are required in the remote host. Machines have a set of attributes pre-defined when the Grid configuration file to be used for executing the application is specified (this is done with the deployment center, see section 3.1.2). These attributes are the ones to be used when defining the constraints of the task. They are:

- OpSys: operating system at the machine.
- Mem: physical memory installed in the machine expressed in Megabytes.
- QueueName: name of the queue in the remote machine queuing system where jobs are going to be executed.
- MachineName: host name of the worker.
- NetKbps: speed of the network interface given in Kilobits per second.
- Arch: processor's architecture.
- NumWorkers: number of jobs that can be run at the same time in the machine.
- GFlops: effective floating point operations that the machine is able to perform in a second, expressed in GigaFlops.
- NCPUs: number of CPUs the machine has.
- SoftNameList: list of software available in the machine.

Initially, the parameter Mem is defined as the physical memory in the machine. However, the semantic of this parameter can be changed when defining the machines that form the Grid. The attributes are a key-value pair, so anything can be defined as a value. The user or a Grid administrator may define in the machines the Mem attribute as virtual memory, or GFlops as theoretical instead of effective. When defining a constraint for a task, the established meaning must be considered.

In order to construct the Expression, the keyword *other* must be used to refer to a machine attribute. Logical and arithmetic operators can be used to build more complex expressions. The SoftNameList is a special case because it contains a list of key values. The built-in function *member* must be used to access its content.

An example of a constraint function is shown in Figure 3.21. It has been specified that, in order to execute a *Dimem* task, a machine with a *powerpc* architecture must be used, and the software named *Dimemas23* must be present in the list of available software (it can refer that the simulator Dimemas version 2.3 must be installed in the machine, or a license must be available to run it). Only the machines that accomplish both requirements will be considered to run a Dimem task. If no constraints should be specified, the value to be returned would be *true*.

```
string Dimem\_constraints(file cfgFile, file traceFile)
{
  return "(other.Arch == \"powerpc\") && (member(\"
     Dimemas23\", other.SoftNameList))";
}
```

Figure 3.21 Constraints specification

The cost of a task is described in a function that will be evaluated at run time. This function must return the estimated execution time of a task in a resource, specified in seconds. The default value 1 is assigned to indicate to the runtime that nothing is specified about the performance of the task in the function (so, the information is not available). We have considered the possibility of calculating the duration of the task as a function of the size of input files and as a function of the performance of the resource used. That is why two functions are defined to assist users in that purpose:

- GS_GFlops(): GigaFlops of the machine that is considered to perform the task. This is extracted as defined at the Grid configuration file.

- GS_Filesize(name): Size of a file in bytes. It is mandatory to use this primitive when a size of a file is needed because the file could be physically in any machine in the Grid.

As the function is evaluated at run time, GS_GFlops will return a different value when different machines are considered to execute a task. The same happens with GS_Filesize, that will return the size of the file in the evaluation moment (this is important if the file has been modified).

The corresponding cost function for the Dimem task is presented in Figure 3.22. The parameters of the cost function are the list of input parameters of the task. In this example, we have previously determined empirically how the size of a trace file affects to the number of operations generated by Dimemas in order to solve the simulation. So, this factor is multiplied by the size of the trace file. And finally the operations that have to be solved are divided by the power in GigaFlops that the machine has. The result is an approximation to the time in seconds that the simulation is going to last.

```
double Dimem_cost(file cfgFile, file traceFile)
{
  double factor, operations, cost;

  factor = 1.0486-e06; //How size of trf
     affects the number of ops generated
  operations = GS_Filesize(traceFile) * factor;
  cost = operations / GS_GFlops();
  return(cost);
}
```

Figure 3.22 Cost specification

The example shows how an estimation could be implemented by the user. However it also exists the possibility of calling inside this function an external program that is able to estimate how much time is going to spend the task in a resource. This way we could use results obtained in research work about run time task prediction, as that topic is out of scope in this thesis.

3.2.6 Programming considerations and tuning

Along this chapter there have been presented several considerations when programming an application with GRID superscalar. This section summarizes all programming considerations in our system. Besides, it also provides some recommendations (non-mandatory) to achieve a better performance when running an application. The main objective of our programming model is to make the Grid transparent to the user, however, some small hints can be helpful to achieve more parallelism in the application.

The file renaming mechanism implemented in the GRIDSs runtime, useful to increase the parallelism of the application, causes some restrictions when operating with files in the master side of the application. We have already seen that GRID superscalar special primitives must be used in the master to open and close files (section 3.2.3) as well as the file descriptors returned by these functions to work with the files. In general, a programmer can never assume that a file has its original name during the execution (but the original name is used when calling to GRIDSs defined IDL functions or the open special primitives). So, the manual renaming or deletion of files that a user can do in the program can cause an incorrect behavior if it is done before GS_Off is called.

When the speculative execution mechanism is used, it has to be considered that the only code that will not be executed in the speculative region in case of exception are the GRID superscalar generated tasks. All the inline code in the master will be executed in all cases (whether the exception comes or not). This is important to be taken into account, because if variables are modified inside the speculative region, these modifications will always be executed.

In the worker side of the application, due again to the renaming mechanism, files are not expected to have their original names, thus the passed parameters in the function must be used to refer to a file. However, temporary files can be safely created and referred with their original names. This is even possible when different instances of the same task could run in the same machine, because the runtime implementation creates a temporal directory in worker hosts to avoid concurrency problems.

In order to tune an application for better performance with GRIDSs, we must take into account that true data dependencies between tasks are unavoidable, and they cause that two tasks cannot execute in parallel. A true dependence is caused when a task wants to read a file (input File) that is generated at a previous task (output File). If the input file is not really necessary to operate in the second task (i.e. it could be some debug information, not needed data), removing it as a parameter in the task definition will avoid data dependencies between these two defined tasks, thus more parallelism will be available in the workflow.

Another suggestion is to completely skip the use of output scalars because the runtime performs automatically a partial barrier to wait for the output scalar to be generated, and this can diminish the parallelism of the application. The rationale for this decision will be detailed in Chapter 4 but it is related with the decision of only considering files as data dependencies in GRIDSs. To avoid the barrier, the scalar could be written to a file.

The GS_Open call used with read or append mode may also cause a partial barrier synchronization in the master code, as the file must be read in that precise moment. If the file is available locally, no barrier is needed because the file can be accessed directly. If the file must be generated from another task, the GS_Open must wait until that previous task finishes to access the file. Again having a synchronization point in the master may decrease the parallelism extraction of the application. Also the wrong usage of GS_Barrier call may also diminish the parallelism, as a full synchronization point is introduced in the master. Again Chapter 4 will describe the details.

3.2.7 Summary

The steps needed to program an application with GRID superscalar are:

- An IDL file that contains the headers of the functions that are going to be run on the Grid must be defined. All files and scalars involved in the computation must be written as parameters, trying to avoid out and inout scalars.

- The master code must be written/changed to call these new defined functions. GS_On must be used at the beginning, GS_Off(0) when the program ends correctly, GS_Off(-1) when an error is detected in the master, and the file management primitives when working with files in the master (it cannot be expected that files have their original names). Avoid the usage of GS_Barrier.

- The worker code must be implemented in a separated file. It must contain the body of the functions defined in the IDL file. Function parameters must be referenced, instead of the expected file names. External binaries must be invoked with GS_System and a possible error code can be reported with gs_result.

In this chapter we have seen many details on the interface, and the advanced possibilities it offers, albeit as we can see in the summary, the minimum steps needed to create a new application with GRIDSs or to adapt an existing one accomplish our objective of simplicity. The internal behavior of the calls in the interface is described in Chapter 4, which describes the GRIDSs runtime in detail.

3.3 Programming comparison

The success criteria of a programming model cannot be measured numerically, as there is no metric to measure how good or bad a programming interface is. The best way to evaluate it is by comparing it with other programming models which follow a similar objective. In this section we compare our programming model with other tools created to assist users to develop and execute their applications for the Grid. The comparison allows us to show if our objective of providing a very simple interface to program has been achieved.

3.3.1 Using Globus

Globus is a Grid middleware which provides basic services to use the Grid. By understanding how these services should be used we may see the importance of hidding the complexity of using a Grid middleware to end users. In this case we present an example of how a simulator could be invoked in several remote machines by using the Globus client interface.

In order to run a remote binary with Globus, both a command line tool may be used, or a program which uses the Globus client interface. If we want to run several instances of the same binary in several different machines in the Grid, we can use both approaches. With the command line tool we need to create a shell script that contains the commands needed, and with the client interface a main program must be created. In both cases the difficult part is related to the creation of a Resource Specification Language (RSL) description of the job to be run, so it is not important which of these two options we select.

We can see in Figure 3.23 an example using the Globus client interface. Whenever users want to invoke a simulator available in a remote machine, they must specify the transfers of the input files required by the simulator, the transfers of results generated, and to erase them from the remote machine once the execution has finished. This must be specified by building an RSL. Moreover, the user must order the job submission to a precise machine, performing the machine selection statically in the program. If more than a single simulation must be run, the user should control how many submissions are done in order to not overload the resources, create a particular RSL for the machine, and specify in every case the machine to be used.

Although the globus_gram_client_job_request used calls are asynchronous to achieve the parallel execution of the binaries, we have not added any state notification handling or error handling in this code for simplicity, so its execution would submit three binaries to three machines and end without waiting for them to finish. The resulting code is Grid-

```
int main()
{
  rsl = "&(executable=/home/user/sim)(arguments=input1.
      txt output1.txt)(file_stage_in=(gsiftp://bscgrid01.
      bsc.es/path/input1.txt /home/user/input1.txt))(
      file_stage_out=/home/user/output1.txt gsiftp://
      bscgrid01.bsc.es/path/output1.txt)(file_clean_up=/
      home/user/input1.txt /home/user/output1.txt)";
  globus_gram_client_job_request(bscgrid02.bsc.es, rsl,
      NULL, NULL);

  rsl = "&(executable=/home/user/sim)(arguments=input2.
      txt output2.txt)(file_stage_in=(gsiftp://bscgrid01.
      bsc.es/path/input2.txt /home/user/input2.txt))(
      file_stage_out=/home/user/output2.txt gsiftp://
      bscgrid01.bsc.es/path/output2.txt)(file_clean_up=/
      home/user/input2.txt /home/user/output2.txt)";
  globus_gram_client_job_request(bscgrid03.bsc.es, rsl,
      NULL, NULL);

  rsl = "&(executable=/home/user/sim)(arguments=input3.
      txt output3.txt)(file_stage_in=(gsiftp://bscgrid01.
      bsc.es/path/input3.txt /home/user/input3.txt))(
      file_stage_out=/home/user/output3.txt gsiftp://
      bscgrid01.bsc.es/path/output3.txt)(file_clean_up=/
      home/user/input3.txt /home/user/output3.txt)";
  globus_gram_client_job_request(bscgrid04.bsc.es, rsl,
      NULL, NULL);
}
```

Figure 3.23 Globus code to submit a binary

aware and in order to handle the parallelism of the application, the code should take into account the notifications coming from tasks, which would add extra complexity to it.

In GRIDSs, a function *sim* must be created as shown in Figure 3.24 where files required are passed as parameters, and the calls to GS_On and GS_Off must be added in the main program. No reference is done to how files must be transferred between resources, or in which machines the simulations are going to be executed. The code is Grid-unaware. Besides, the parallel execution of the application is handled by the GRIDSs runtime implicitly without any effort for the user.

```
void sim(File input, File output)
{
  command = "/home/user/sim " + input + ' ' + output;
  gs_result = GS_System(command);
}

int main()
{
  GS_On();
  sim("/path/input1.txt", "/path/output1.txt");
  sim("/path/input2.txt", "/path/output2.txt");
  sim("/path/input3.txt", "/path/output3.txt");
  GS_Off(0);
}
```

Figure 3.24 GRIDSs code to submit a binary

3.3.2 Using DAGMan

Condor DAGMan [30] is an environment which allows to define workflows to be executed in the Grid. It has been already introduced in Section 2.1.1. The definition of the workflow is made in the DAG input file, an input passed to DAGMan. There exists a particular syntax where dependencies between jobs are explicitly specified. To present an example using DAGMan, we are going to focus in the keywords JOB and PARENT. The JOB keyword specifies a job to be managed by Condor where a job name and a Condor submit description file must be passed. This submit description file describes the binary to be run and other submission details, but nothing about where the job must run needs to be specified (Grid-unaware).

Figure 3.25 presents a simple workflow structure defined with DAGMan, known as the diamond example because of the generated graph. In this case there is a source node A, from where two edges point to nodes B and C. From B and C there is an out edge pointing to node D. We can see how the Condor submit description file is passed, and

```
JOB A A.condor
JOB B B.condor
JOB C C.condor
JOB D D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Figure 3.25 DAGMan example to build a workflow

```
int main()
{
  GS_On();
  task_A(f1, f2, f3);
  task_B(f2, f4);
  task_C(f3, f5);
  task_D(f4, f6);
  GS_Off(0);
}
```

Figure 3.26 Same workflow built using GRIDSs

how the dependencies are described. Using GRIDSs to generate the same workflow, four different functions should be specified to define the four tasks performed, as it is shown in Figure 3.26. The GS_On and GS_Off calls are used, as they are mandatory for every GRIDSs program. The files needed in every task are passed as parameters in the calls, in contrast to what is done in DAGMan, where these files are specified in the several Condor submit description files.

Although both approaches are Grid-unaware, the main difference between them is that with DAGMan the dependencies between jobs are explicitly described in the DAG input file, while in GRIDSs these dependencies are implicitly extracted from the actual parameters of the tasks. Moreover, if we want to generate the same DAG a thousand times, in DAGMan we should repeat the content of this file a thousand times, while in GRIDSs only a loop calling a thousand times to the defined functions should be used. Not only loop constructs can be used to generate the workflow, but also conditional structures provided by the programming language (if, switch, ...), thus a different workflow may be generated for different executions of the application.

In addition, intermediate files can reuse the same name because the runtime will apply renaming to them (see Chapter 4). This is an important advantage, since the developer does not have to manage a large number of intermediate files, as it is done by the system. In this sense we can say that GRIDSs is more efficient than DAGMan from the programming point of view, because bigger workflows can be generated easier.

3.3.3 Using Ninf-G

The Ninf-G programming model [97] provides a framework for programming on the Grid and it is a reference implementation of the GridRPC API [83]. GridRPC includes two types of calls, synchronous and asynchronous. While synchronous calls allow to implement a sequential algorithm, they do not allow to exploit the concurrency of the application, as only a single call may be executed in a precise moment. Asynchronous calls allow to implement parallelism in the application, but the parallelism must be explicitly described in the code.

Figure 3.27 shows an example of how asynchronous calls are used in Ninf-G. In order to execute the *foo* function in two machines in the Grid (A and B), a handle object must be created specifying the machines to be used. After that, the work must be splitted between the two machines, thus the tasks containing odd input and output files will be executed in machine A and the even ones in machine B. In this case the user is not only specifying in which machines the functions must be run, but also how the work is splitted among them. This means that the programming model is Grid-aware and that the parallelism is explicitly described. Moreover, the grpc_wait_all call blocks until all function calls have finished, so the synchronization is also explicitly done by the user.

```
int main()
{
  grpc_initialize("config_file");
  grpc_object_handle_init_np("A", &A_h, "class");
  grpc_object_handle_init_np("B", &B_h," class");
  for(i = 0; i < 25; i++)
  {
    grpc_invoke_async_np(A_h,"foo",&sid, f_in[2*i],
        f_out[2*i]);
    grpc_invoke_async_np(B_h,"foo",&sid, f_in[2*i+1],
        f_out[2*i+1]);
    grpc_wait_all();
  }
  grpc_object_handle_destruct_np(&A_h);
  grpc_object_handle_destruct_np(&B_h);
  grpc_finalize();
}
```

Figure 3.27 Ninf-G usage example

When implementing the same application in GRIDSs (see Figure 3.28) only the GS_On and GS_Off calls must be added, as calls to functions in GRIDSs are already asynchronous. In addition, nothing is specified in the code about where a function must run, thus keeping the programming model unaware of the underlying Grid. As the GRID superscalar runtime will be in charge of deciding where a task must be executed, no special indexing must be used to split the work among the available machines, at it was done with Ninf-G. In both cases an IDL file describing the headers of the functions is used to know which files must be sent to or received from a machine when a task is executed.

```
int main()
{
  GS_On();
  for(i = 0; i < 50; i++)
    foo(f_in[i], f_out[i]);
  GS_Off(0);
}
```

Figure 3.28 Same application using GRIDSs

3.3.4 Using VDL

The Chimera's VDL language [35] is used to define abstract workflows, as it has been briefly mentioned in Section 2.1.2 when describing Pegasus. The "abstract" word added to the workflow means that no machine is specified in the workflow, only the task which is an application invocation. Thus, the abstract workflow can be converted to a specific one when it is going to be executed in a particular Grid (this is what Pegasus does), adding the data transfers if needed.

There are two main constructs when creating a workflow with VDL. The first one is a transformation (denoted TR), which is an abstract template for invoking a binary. Comparing it to GRID superscalar it may be similar to defining a function to be run in the Grid, but the function can only contain a binary invocation. The second main construct is a derivation (denoted DR), which corresponds to a materialization of a transformation. It is how the transformation is made specific by passing the parameters to be used in the transformation. Following with the comparison with GRID superscalar, the derivation would be the call to the function in the main program.

We present an example of a VDL workflow in Figure 3.29. We can see that between these two derivations there exists a data dependence (trans2 cannot be executed until

```
DV trans1( a2=@{output:tmp.0}, a1=@{input:filein.0} );
DV trans2( a2=@{output:fileout.0}, a1=@{input:tmp.0} );

DV trans1( a2=@{output:tmp.1}, a1=@{input:filein.1} );
DV trans2( a2=@{output:fileout.1}, a1=@{input:tmp.1} );

...

DV trans1( a2=@{output:tmp.999}, a1=@{input:filein.999} );
DV trans2( a2=@{output:fileout.999}, a1=@{input:tmp.999} );
```

Figure 3.29 VDL workflow application

trans1 generates the data) which is automatically detected by Chimera. Besides, the direction of the parameters used must be specified in every derivation, which makes the interface look more complex. Another problem is that each derivation must be specified in a new line: no constructs such as loops or conditionals are provided to create a set of derivations. This means that the size of the VDL program grows proportionally with the size of the workflow. So, if we want to call trans1 and trans2 with a different input file a thousand times, two thousand lines should be added to the program.

In contrast, as GRID superscalar is based on an imperative programming language to generate the workflow, a loop can be easily used to call to the transformations all times that should be needed. This is shown in Figure 3.30. So, in both cases the executions can be performed in parallel using the Grid, but in GRIDSs the interface allows to build very big workflows in a simpler manner. Both interfaces are Grid-unaware and the parallelism is implicitly exploited, however, if part of the application is not very computation demanding, in GRIDSs it may be executed locally in the master code, while in VDL the application must be composed of binaries which will run in the Grid.

```
int main()
{
  GS_On();
  for(i = 0; i < 1000; i++)
  {
    tmp = "tmp." + i; filein = "filein." + i;
    fileout = "fileout." + i;
    trans1(tmp, filein);
    trans2(fileout, tmp);
  }
  GS_Off(0);
}
```

Figure 3.30 Workflow application generated with GRIDSs

Chapter 4

GRID superscalar runtime

The GRID superscalar runtime is the other main component of the GRID superscalar programming environment. We have already presented in previous chapter the programming interface, corresponding to the static phase of the application (when the application is programmed and deployed). The runtime corresponds to the dynamic phase of the application: the execution. The design of the runtime is focused on one hand on implementing the functionality provided by the programming interface, and on the other hand on implicitly exploiting the existing parallelism in the programmed applications. We have seen in the description of the programming interface that it does not provide any calls to express parallelism in the code, as we want it to follow a sequential programming paradigm in order to be very simple. This means that the parallelism must be extracted from the application automatically, and the runtime will be in charge of this.

The final objective of executing the application in parallel is to speed up its execution time by means of using the resources available in the Grid. Therefore, the success criteria of our runtime depends on the reduction achieved of an application's total execution time. Anyway, we will see that the achievable parallelism of an application depends on its implementation (more specifically, on its internal data dependencies).

A main requirement for our runtime is the usage of a Grid middleware, which provides the basic services to use the Grid infrastructure: job management, data transfer and security. The job management is needed to execute the workers of the application remotely, the data transfer to send input and output files between machines, and the security to allow the authentication an authorization in the machines. Our runtime is responsible of using the Grid services and hide them to end users, together with the programming interface. We selected initially the Globus Toolkit [33] as a reference middleware to implement the runtime, but we are not bound to Globus, as we have been able to easily port the runtime to other middlewares.

The first section of this chapter describes in detail the main idea applied in the runtime: apply our knowledge from the computer architecture world to a completely different environment, such as the Grid programming environments. An automatic data dependency detection and a renaming mechanism enable to execute parts of the application in parallel.

The second section presents more developments made with the implementation of the runtime. First we show how the basic functionality of the programming model has been implemented and then we present how the task scheduling is performed in the runtime considering three different point of views: the task dependency graph, the brokering of resources and the file locality exploitation. At the end of the section it is described how the workers can communicate with the master, how the application finishes its execution, and how some pre-defined environment variables allow to tune the behavior of the runtime without the need of recompiling neither the runtime nor the application.

The last section shows the main experiments made with the runtime. First we present a deep analysis of the runtime made with the NAS Grid Benchmarks, and then we show a simple example and some real use cases of GRIDSs to study the scalability of the implemented applications. Although in these real cases the main work has been done by other authors, we believe they are very important to show the achievements made by our environment.

4.1 Runtime scientific contributions

In order to achieve the goal of speeding up the execution of a Grid application, we have adapted ideas coming from the computer architecture world, more precisely the superscalar processor design. Our vision is that a processor internally is very similar to the Grid: they both have computing elements, storage elements, and a network that interconnects all of them. The main difference between the Grid and a processor is the timescale in which everything happens: while in a processor storing data into memory or running an instruction can be measured in nanoseconds, in the Grid transferring data to or running a binary in a remote machine can last seconds, minutes, or even hours.

As we have found this similarity between processors and the Grid, we want to use superscalar processor techniques in a completely different environment such as the Grid. In a superscalar processor, the dispatcher reads instructions from memory and decides which ones can be run in parallel, dispatching them to redundant functional units contained inside a single processor. Therefore, a superscalar processor can be envisioned having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread. In the Grid, our runtime will decide which tasks can run in parallel (the IDL tasks defined by users, as seen in Chapter 3), dispatching them to the available remote machines in the Grid, thus achieving the parallel execution of tasks from a single thread program (the master).

Two key mechanisms are applied in our runtime to implement the parallel execution of tasks. The first one is the automatic data dependency detection, and the second is the parameter renaming. They both will be described in detail in next sections.

4.1.1 Automatic data dependency detection

As in superscalar processors, in GRIDSs a data dependency analysis is performed between all tasks that form the application to exploit its parallelism. A graph is used in order to represent the tasks (nodes in the graph) and their data dependencies (directed edges). A node is an instance of one of the operations defined in the IDL file to be executed in the Grid. The edges in the graph define a relative execution order that must be respected when scheduling the tasks in the Grid. For that reason, this graph is also called *workflow*.

Task generation is done based on the execution of the main program (the master). As it has been described previously, this program is sequentially run in a single thread of execution and this defines a *sequential order* between tasks. During the run of the program, every time an IDL defined operation or a GRIDSs API call is found, an interaction with the runtime is performed. In the case of IDL operations, the Execute call is invoked and a task is generated. At this step, there is also a data dependency analysis done between the new created task and all the previous generated tasks during the execution of the master. The generated task is inserted in the task dependency graph, and in most of cases it will not be executed immediately but will remain pending for execution. However, once the task has been inserted in the graph, the runtime returns control to the main program that will continue with the master execution, thus continuing with the task generation. Tasks are sequentially generated, but thanks to the data dependency analysis, they can be executed in an out-of-order manner while data dependencies are maintained. So, a tasks generated later in the sequence can be run before, or even at the same time of a task generated earlier when no data dependencies exist between them.

Whenever the master has a synchronization point, the generation of tasks is stopped, as the master does not continue its execution. Synchronization points in the code can be caused by calling to GS_Off, GS_Barrier, GS_Speculative_End and a GS_Open or GS_FOpen where a file must be read. The master is also stopped if output scalars are defined in a task. All these situations were previously introduced in Section 3.2.6 as situations to be avoided, but now we can understand why they influence in the parallelism exploitation of the application. Stopping the generation of tasks is not good because future tasks that potentially may have been executed in parallel will not be created if the master program is stopped. Thus if the generated graph before the synchronization point does not have enough degree of parallelism to use all available machines in the Grid until the synchronization is solved, we are losing the opportunity of achieving a better speed up of the application. Nevertheless, depending on the application, the synchronization is mandatory and so it may not mean that a better performance may have been achieved (i.e. GS_Off is used to finish, so nothing more is going to be executed in the Grid. Or GS_Speculative_End, which waits for all speculative tasks). When the synchronization has finished, the generation of tasks continues.

The data dependencies considered between tasks are the ones found when analyzing the files generated or consumed by a task. The reason for only considering files in this analysis is because of the type of applications our system targets, as it has been described in section 3.1.1. A data dependency analysis considering also scalar parameters of a task would not provide any big gain in the type of applications we focus and would require a big effort in developing a new compiler. This compiler should be able to perform a data dependency analysis between any instruction in the code and any call to a GRIDSs function, because in order to know in which instruction an output scalar is used, every single instruction in the main code should be analyzed to see if the output scalar is referenced. Besides, any scalar variable can be easily transformed in a file parameter by writing its value to a file, so considering its data dependencies.

There are several types of data dependencies which may be found between two tasks [47]:

- Read after Write (RaW): exists when a task reads a file that is written by a previous one. The second task cannot continue until the file needed from the first task is generated. An example is presented in Figure 4.1. In this case, the result of task *filter* is written in bh_tmp.cfg, and task *dimemas_funct* uses that file as input. Therefore, task dimemas_funct must be executed after task filter since it needs an output of this task.

- Write after Read (WaR): exists when a task writes a file that is read by a previous one. If the second task overwrites the file, the first one would read an incorrect content. We give an example of this type of dependency in Figure 4.2. In this case, the problem is with the parameter dim_out.txt, which is read by task *extract* and written by task *dimemas_funct*. Initially, if the sequential execution flow is followed, no problem should arise. However, if dimemas_funct is executed before task extract it may overwrite some data needed by task extract.

- Write after Write (WaW): exists when a task writes a file that is also written by a previous one. If the write order between these two tasks is not kept, the content left in the file for upcoming read operations would not be correct. Figure 4.3 shows an example of a WaW dependency. Both tasks named *filter*, write in the same file bh_tmp.cfg. A problem with this data dependency may arise if the first instance of task filter is executed before the second one. Although the execution of both tasks will not be affected, if there is any task that later reads bh_tmp.cfg from second function, instead it would read the resulting file from the first function.

```
filter("bh.cfg", L, BW, "bh_tmp.cfg");
dimemas_funct("bh_tmp.cfg", "trace.trf", "dim_out.txt");
```

Figure 4.1 RaW dependency example

```
extract("bh_tmp.cfg", "dim_out.txt", "final_result.txt");
dimemas_funct ("bh_tmp.cfg", "trace.trf", "dim_out.txt");
```

Figure 4.2 WaR dependency example

```
filter("bh1.cfg", L1, BW1, "bh_tmp.cfg");
filter("bh2.cfg", L2, BW2, "bh_tmp.cfg");
```

Figure 4.3 WaW dependency example

It must be clarified that, if the code is executed sequentially, there is no problem in any of these examples. But when parallelism is exploited, the dependencies represent an order restriction between the tasks. In order to detect a RaW dependency, the input files of the created task must be compared with the output files of all existing tasks. The WaR dependency is detected with the opposite operation: comparing all output files defined in the created task with the input files of previous existing tasks. And the WaW dependency is detected by comparing all output files, both from the created task and the existing ones. Because a task can have as many input or output files as needed, several types of data dependencies can exist between two tasks. Nevertheless, we will see in section 4.1.2 that RaW dependencies are the only ones that are unavoidable, so whenever a RaW dependency is detected, there is no need of checking for WaR and WaW dependencies between two tasks, as the RaW dependency already makes mandatory the sequential execution of both tasks. This reduces the complexity of comparing the parameters of current task against all previous generated tasks.

We can ensure that the resulting graph of a data dependency analysis does not have any cycles, neither between new tasks pointing to old ones nor between a task itself. The former is due to the definition of the dependencies, which is made to keep the sequential order of the application defined by the main program in the master. The latter is because no order needs to be kept in a task itself, as the task is the atomic unit of execution in the system. Therefore, graphs that define data dependencies are also known as Directed Acyclic Graphs (DAG).

During the execution of the master, any call to the API file management primitives (GS_Open, ...) is also analyzed to check data dependencies. The open primitives are translated to a task, with the particularity that this task will be executed locally. Depending on the mode specified in the open (read, write or append), the task will be created with an input file (read), an output file (write), or an input and output file (append) as parameters.

Therefore, when a file is open for reading or appending data, a RaW dependency may be created if the file is the result of an existing task. If the file is opened for writing or appending data, a WaR or WaW dependency may be detected if a previous task reads or writes in the file, respectively. Besides, the execution of the master program cannot continue until the dependency is solved, as following operations will probably access the file requested to be open. Next section explains how renaming techniques solve this problem for WaR and WaW dependencies.

From the resulting task graph, the GRID superscalar runtime is able to exploit the parallelism, by sending tasks that do not have any dependency between them (no path exists between them in the task dependency graph) to remote hosts in the Grid. The program will execute correctly if all types of task dependencies are respected. This way, our runtime extracts and exploits the implicit parallelism inside the application.

4.1.2 Parameter renaming

From the three data dependencies introduced in previous section, the RaW dependency is unavoidable, as nothing can be done in order to try to execute both tasks in the dependency with a different order than the one specified. For that reason this dependency is also called *true dependency*. However, WaR and WaW dependencies can be avoided using a parameter renaming mechanism.

Register renaming is a technique used in superscalar processors in order to remove WaR and WaW dependencies between instructions [47]. That is why those dependencies are also called *false dependencies* because they can be avoided with the mentioned renaming technique. The renaming removes the implicit order between both tasks, therefore the graph describing the data dependencies between instructions has less edges, which is important in order to increase the possible parallelism in the application. In the GRID superscalar runtime this mechanism is also included, which in this case renames files instead of registers in order to avoid data dependencies between tasks.

The renaming is just, as the word suggests, the change of the parameter's name. Considering the examples presented in previous section (Figures 4.1, 4.2 and 4.3), we now present the same cases for WaR and WaW dependencies but with a renamed parameter. In Figure 4.4, instead of using the same dim_out.txt as parameter, if the system renames the file parameter for the dimemas_funct task, then the WaR dependency would disappear, because dimemas_funct can safely write in the file dim_out.txt_RENAMED without overwriting the content of dim_out.txt. Thus, both tasks can now be executed in any order or even at the same time. Similarly, in Figure 4.5 the parameter bh_tmp.cfg has been renamed in the second call to the filter function to bh_tmp.cfg_RENAMED, so again both tasks can be executed in any order.

```
extract("bh_tmp.cfg", "dim_out.txt", "final_result.txt");
dimemas_funct ("bh_tmp.cfg", "trace.trf", "dim_out.
   txt_RENAMED");
```

Figure 4.4 WaR dependency renaming example

```
filter("bh1.cfg", L1, BW1, "bh_tmp.cfg");
filter("bh2.cfg", L2, BW2, "bh_tmp.cfg_RENAMED");
```

Figure 4.5 WaW dependency renaming example

An example of how the data dependency graph is reduced by the renaming technique is shown in Figure 4.6. The file f1 is an output in task T1 and an input in T2, which defines a true data dependency in the first iteration, but in subsequent iterations there exist WaR and WaW dependencies that can be avoided. We can see that removing these dependencies the graph is transformed from a totally sequential graph to a parallel graph. In summary, the most important achievement of the renaming is to increase the task graph degree of concurrency, in order to enable more tasks to be executed in parallel when running the application in the Grid.

```
While (!end_condition())
{
   T1 (…,…, "f1");
   T2 ("f1", …, …);
   T3 (…,…,…);
}
```

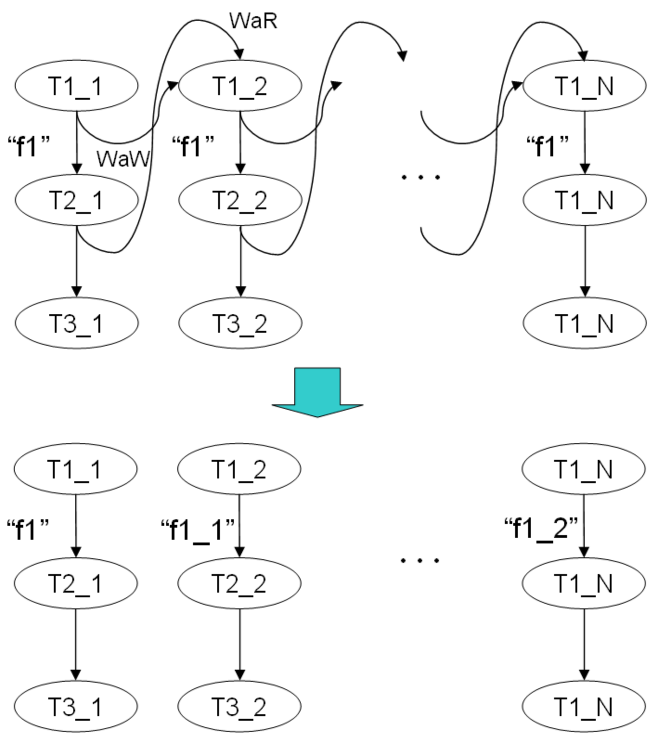


Figure 4.6 Example of a data dependency graph optimization with file renaming

The renaming must be taken into account in the data dependency analysis. To rename a file in a task in the end means to have a new co-existing version of that file, as two different processes will write in two different files that initially were the same. It is important to mention that future calls in the algorithm using a file which has been renamed, must be translated to the new name generated in order to reference the correct version of the file. For instance, in Figure 4.5, any new reference to bh_tmp.cfg must be translated to bh_tmp.cfg_RENAMED to keep results of the algorithm correct. Thus, when a new task is generated, and before doing the data dependency analysis, the runtime internally substitutes the filenames of the parameters referenced in a task by the last renamed filenames.

Internally, GRID superscalar handles the renaming by means of two hash tables: one that, when given the original filename, returns the name of the last renamed filename and a second one that, when given a renamed filename, returns the original name. A original filename may have several renamed filenames during the execution of the whole application, but a renamed filename has always only one corresponding original filename. When generating a new task, the only renamed filename that can be used is the last one, as our programming model maintains the same behavior as if the application has been executed sequentially. We will see that, because of this, we will be able to implement mechanisms to remove intermediate versions of a file when they are not going to be used anymore (see Section 4.2.4).

The runtime has different behaviors depending on the type of dependency. When a RaW dependency is found, the data dependency is directly added in the graph, as it is unavoidable. When a WaR or WaW dependency is found, a new name for the output file that causes the data dependency is generated. Data structures are updated and both tasks could eventually be run in parallel. No dependency is added to the graph. In the end, the graph only contains RaW dependencies, as all WaR and WaW dependencies must have been removed with the renaming.

The GRID superscalar file management primitives (GS_Open, ...) may also cause renaming in file parameters. Whenever a WaR or WaW dependency is detected between the open task and previous tasks (this may happen when the task has an output file, thus if open mode is specified as append or write), a renaming is introduced. Moreover, future generated tasks will also check dependencies with tasks generated because of a call to GS_Open, so more renamings could be introduced later.

There is an extra benefit in adding a renaming for a call to open a file, because any dependency detected must implement a barrier synchronization in the open call to respect the data dependency, as the code following the open is probably going to perform operations to the file (using the file descriptor returned as an output scalar). This means that the master code cannot continue executing until the data dependency is solved. When the open causes a RaW dependency, the barrier cannot be avoided (data generated by another task must be read). However, if the dependencies detected are WaR or WaW, a renaming allows to directly write in a new created file (the rename), therefore allowing the master main code to continue its execution, generating more tasks that potentially can be run in parallel.

When the execution of the main program finishes, even if it is a correct execution of the whole application or when there is an error and the execution is stopped, all the renamings made to files are undone by the runtime. This way the renaming technique applied to files keeps hidden to users, which is one of our main objectives. Besides, in order to make users able to check for results during the execution of the application, other mechanisms are implemented to undo the renaming during the execution (described in Section 5.2.1).

4.2 Runtime developments

4.2.1 Basic functionality

This section describes how the basic functionality designed in our programming model is implemented, as the objective of the GRID superscalar runtime is not only to achieve the parallel execution of the programmed applications, but also to implement the functionality provided by the programming interface.

Job submission

The basic functionality provided by the runtime is to run tasks into remote machines, which is enabled by the usage of a Grid middleware that allows the usage of distributed resources (CPUs, disks, ...). Several services provided by a Grid middleware are required in order to run a GRID superscalar task in a remote machine (which is also known as a job). More precisely, our runtime uses directly the job management service (to submit, monitor or cancel jobs) and the file transfer service (to move data between machines). However, other services also provided by the middleware are passively used, such as the security service (authorization and authentication of the user in the remote environment).

Submitting a task to a resource requires four stages: first send the input data files required for the task from their original location to the host where the task is going to be executed, second start the job by starting a binary in the remote machine, third detect when the job has finished its execution, and fourth collect the results of the job. For every single task, our runtime is in charge of performing these Grid related actions totally transparent to the user. This keeps the user away of knowing any details about how the Grid middleware must be used, thus achieving our objective of keeping the Grid as transparent as possible for the user.

The selected middleware to implement our runtime on top of it has been the Globus Toolkit [33]. This decision is justified with the wide usage of this middleware by many other tools for the Grid. The Globus Toolkit has emerged as a standard *de facto* in the Grid community. It provides a set of basic services divided into four main categories: security, data management, execution management and information services. This set of services, and the API provided to use them, allows enough flexibility to implement the GRID superscalar runtime with them.

Selecting Globus as the reference middleware may have caused that some implementation decisions are bound to it (i.e. using the Resource Specification Language (RSL) to describe how a job must be run in the Grid). However, the core of the GRID superscalar is independent of the Grid middleware, and therefore future versions of GRID superscalar could use other middlewares. This has been demonstrated in work done mainly by other members in our research group adapting GRIDSs to run on top of other middlewares: Ninf-G [97] and ssh/scp (see [5] and [9]).

When a task is submitted to the Grid, the first thing to be done is to provide the inputs it needs in order to run. In GRIDSs, the inputs which are files are transferred using the file transfer service from their initial location to the destination machine. The scalar inputs are provided as arguments to be passed to the command line when the job is run (encoded in base64 [36] format). Once files are transferred, a temporal directory must be created in order to ensure the correct execution of the task. This is due to the possibility of working with temporary files in the task, because if several tasks are executed in the same machine working with the same temporary files, the consistency of these temporary files could not be ensured. When the temporal directory is created, the execution management service starts the binary in the remote machine, and checks periodically the state of the running process to notify about state changes to our runtime. Whenever the binary finishes, a notification is sent to the GRIDSs runtime. After that, in the end of the process, the task file results may be sent to the master using the file transfer service.

The task submission is done with the execution management service provided by Globus. Some of the steps needed to perform in order to submit a task can be all specified together with the Resource Specification Language (RSL). This language implemented in Globus allows to specify which files have to be transferred to the destination machine, which scalar parameters must be passed as command line arguments, when a new temporal directory must be created, which binary must be invoked in the remote machine, the working directory, which files must be erased after execution, and which files are results to be transferred to the master machine. Inside our runtime, for every task, an RSL is built in order to submit the task to the Grid with the Globus API. This is an implementation detail bound to Globus, however the standards follow similar approaches (JSDL in SAGA [43] and job templates in DRMAA [19]), thus it does not limit the adaptation of our system to other middlewares.

An important part of the task lifecycle in the Grid middleware is how the state change of the job is handled. Globus provides a *callback system* to receive notifications from submitted jobs. This monitoring system is based on notifications sent from the job submission service to the client API. A TCP port must be opened at the beginning of the execution, and provided in the task submit in order to receive the notifications. A callback function can be programmed, so whenever a notification of a job state change is received, the callback function is invoked from Globus in order to deal with the state change in the client (the GRIDSs runtime is the Globus client).

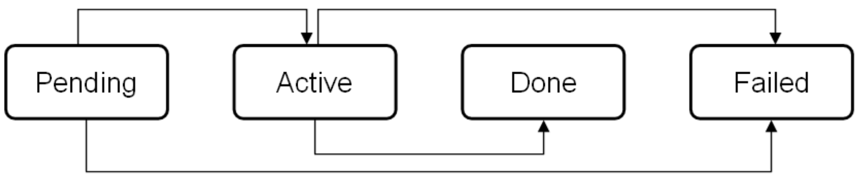


Figure 4.7 Simplified job state diagram

Using this callback system in GRIDSs means that our runtime is event based, as events (state changes) are received and treated immediately. This is opposite to a poll based system, where the runtime would be in charge of polling for the status of every submitted task. In our case, both approaches could have been implemented. The callback system in Globus has a small limitation, because no job submissions can be made inside it. Anyway, the data structures in the runtime are updated depending on the notification received, and the job submits are performed outside this function.

The state notifications related to a job that our runtime is interested in are:

- Pending: the job is waiting in a queue system (a local resource manager) in the remote machine in order to be started. In this state we also know that the transfer of input files has been completed.
- Active: the binary has started to run in the remote resource. In this state we also know that the transfer of input files has been completed.
- Done: the job has finished and the results have been transferred to where it was specified.
- Failed: there has been a middleware error during the execution. An error code is provided to find out more about the error.

The state diagram of a job is detailed in Figure 4.7, with the corresponding transitions available. Note that not all states supported in Globus are represented, as we have only detailed the states important for the GRIDSs runtime.

As a summary of the basic functionality provided by the GRID superscalar runtime, and to understand better its behavior, we provide two UML [103] sequence diagrams. Figure 4.8 describes the interactions from the master point of view. The diagram represents a very simple main program, with a single call to tasks T1 and T2. The runtime is implemented as a dynamic library linked with user's application. The access to the library is performed explicitly from the user code, using the GRID superscalar primitives

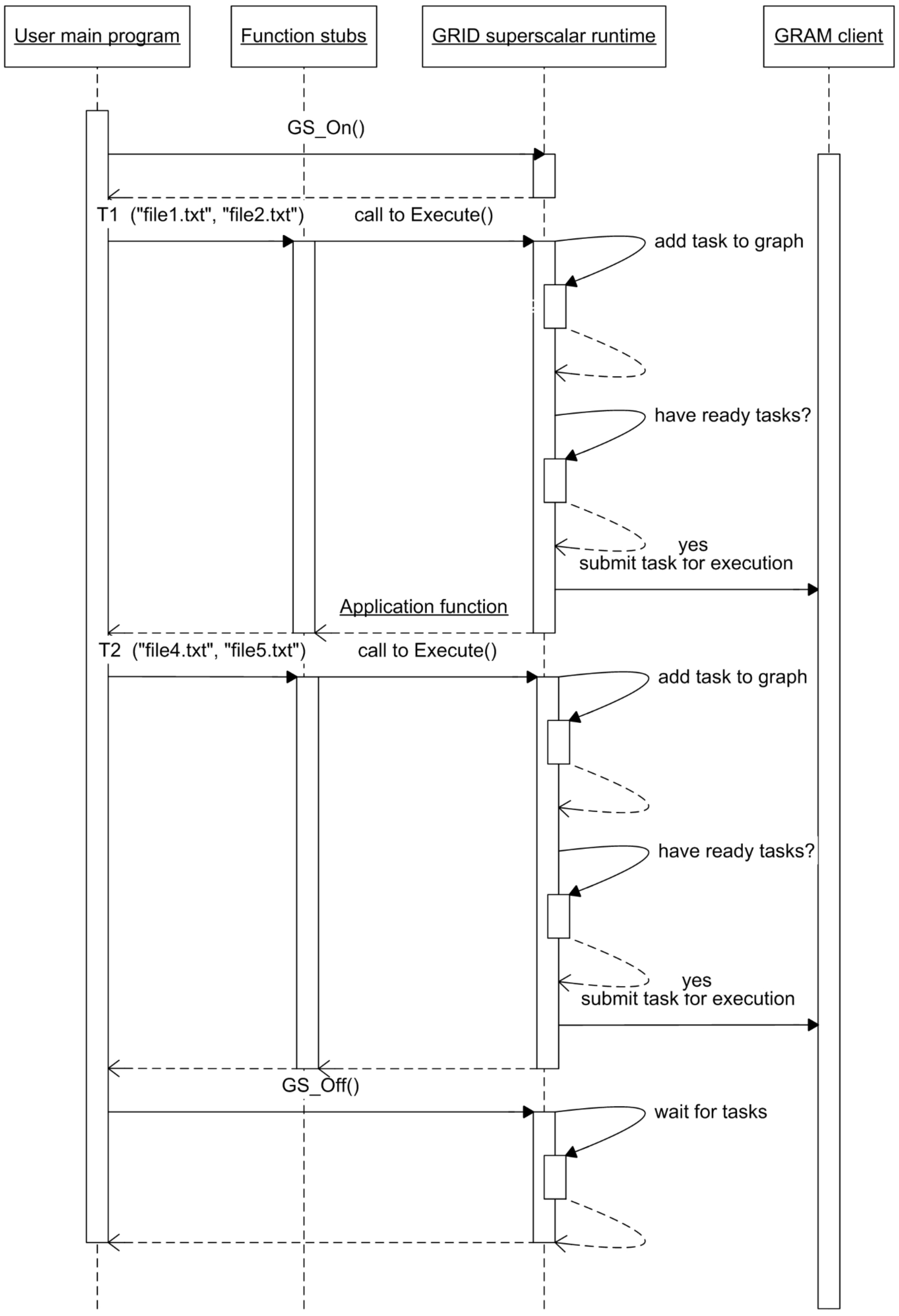


Figure 4.8 UML sequence diagram of the behavior of the GRID superscalar runtime (master side)

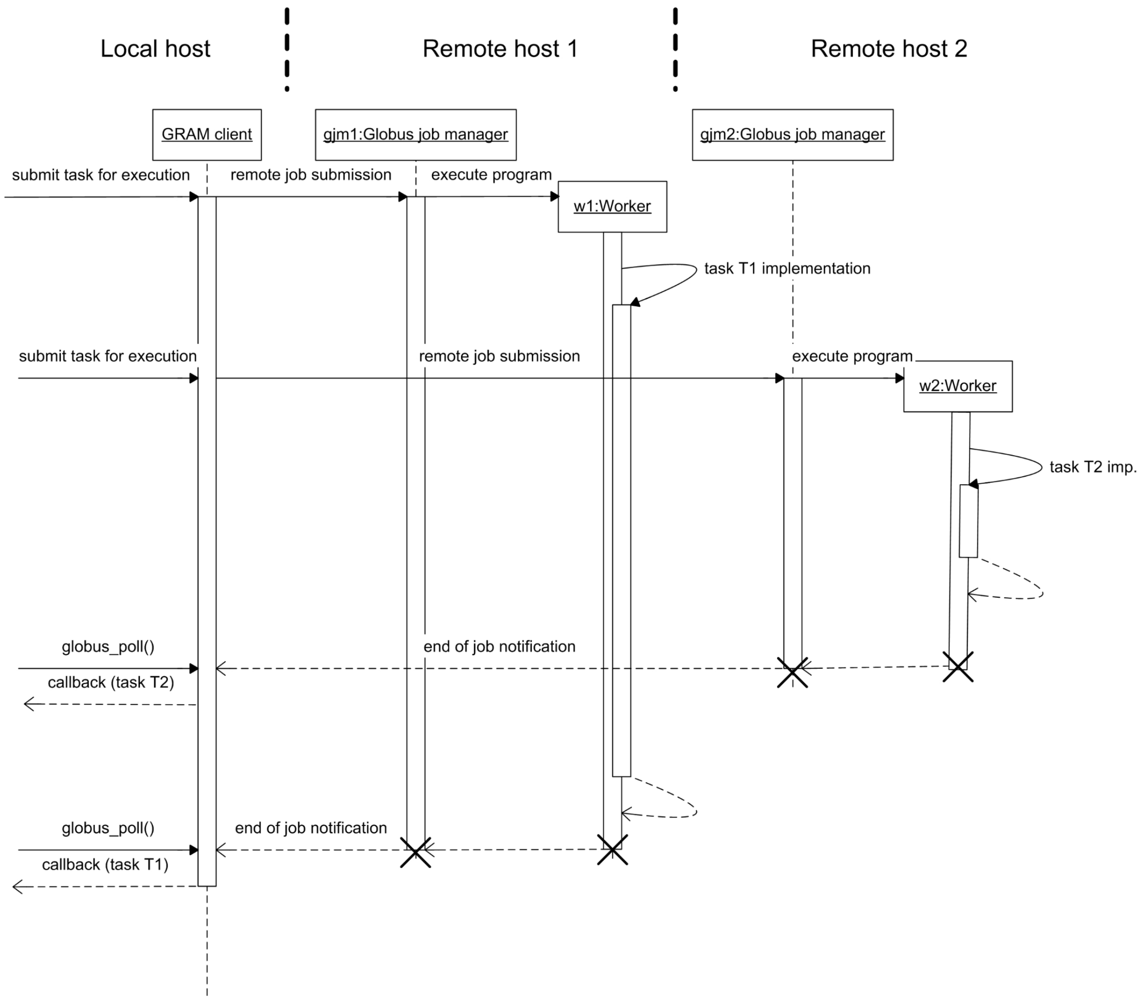


Figure 4.9 UML sequence diagram of the behavior of the GRID superscalar runtime (worker side)

(GS_Open, GS_Barrier, ...) and in a transparent way to the user, through the calls to the basic primitive *Execute* which are performed from the generated stubs.

Figure 4.9 shows the UML sequence diagram of the behavior of the GRID superscalar run-time from the worker point of view, where we can see a request to the Globus execution management service (GRAM) and how it runs the binaries of the example, sending a notification back to the master and calling the callback function to treat the state change.

API implementation

In next lines we are going to describe how the calls defined in the GRID superscalar API are internally implemented to achieve the behavior previously described. These calls where previously described in Section 3.2.3.

The first one is GS_On. In this call, apart from creating all data structures needed in the runtime (dependency graph, hash tables for names, ...), there is also the need of reading all environment variables that may be introduced by the user or even a Grid administrator in order to tune the behavior of the runtime. and read the configuration file which contains the details of the machines which form the Grid. The environment variables are detailed in Appendix A and the configuration file format in Section 4.2.3. It also performs initialization calls to the Grid middleware in order to use it.

The GS_Off call is in charge of waiting for all previous generated tasks to finish by using a GS_Barrier call. Once they are finished, it recovers all result files that may still be located in remote workers, and erases all data files that have been in the workers during the execution, to leave the machines as they were before the execution of the application. Finally, it restores the original names of renamed files during the execution, and stops the usage of the Grid middleware services.

In the case of the GS_Open and GS_FOpen calls we have seen that, in order to check their data dependencies a task is created. This is done with a call to the Execute function, using a special type of operation identifier, so when the task must be executed, the runtime knows it must be run locally. The file parameter passed to Execute depends on the mode the file has been opened: in read mode an input file, in write mode an output file, and in append mode an input/output file. In all cases an output scalar is defined, which contains the file descriptor result of performing the open system call. This descriptor is returned directly to the user in GS_Open, and previously translated to a file pointer type in GS_FOpen. When the local task is executed, it is checked if the file is present in the master machine, or must be recovered from a worker machine, by using the file transfer service.

GS_Close and GS_FClose can be understood as a end notification of the local task. So, the local task is removed from GRIDSs data structures, and the file corresponding to the descriptor passed as a parameter is closed.

Whenever GS_Barrier is called, either in the main program explicitly by the user or internally in the runtime, the function does not return until all generated tasks at that moment have finished. The only thing this call does is wait for notifications of tasks so they can be treated in the corresponding callback function.

The GS_System call, defined to be used in the worker, translates the path inside the command line passed as a parameter because a temporal directory is created to avoid concurrency problems with files during the execution of the job. This way, the binary specified by the user will be correctly located. If the path is an absolute path, it is not modified. It returns the result of calling the *system* call in the C standard library.

Finally, the GS_Speculative_End and GS_Throw are explained together, as they both belong to the speculative task execution mechanism. Whenever the GS_Speculative_End call is used at the master, there is a conditional barrier that waits until all previous tasks finish or an exception is detected in a worker. If no exception is notified, the execution of the master continues normally after the speculative region, so the behavior is like any other GRID superscalar program where all tasks are executed. If an exception is notified, all tasks generated after the one that has risen the exception and until the GS_Speculative_End call must be discarded, as it is considered that the exception task is the last valid task. If the task to be discarded is running, we must cancel its execution, if it is waiting, it will be erased, and if it has been already executed, their results are discarded.

When no more speculative tasks are running, the tasks are deleted from the runtime as if they have not existed (in the dependency graph and other internal structures). Moreover, the possible renaming introduced is undone until the last valid task, which contains the last valid result files. At this point, if a function has been specified in the GS_Speculative_End call, it is invoked. After that, the execution of the master program continues normally.

The GS_Throw primitive called in the worker uses the gs_result parameter introduced also in Section 3.2.3. This parameter is used to specify an error code in a task to the master. A special code is specified to notify that a GS_Throw has been called, and the function that was being executed returns at that point.

4.2.2 Task scheduling: the task dependency graph

From all the tasks generated in a GRID superscalar application, we have to decide which ones will be submitted to the Grid. In order to make that decision, the task dependency graph resulting both from analyzing data dependencies between tasks and with false data dependencies removed is essential. The graph represents a relative order between tasks that cannot be ignored in any case, thus whatever task scheduling policy that wants to be proposed in our runtime must use the graph as a basic input.

As the graph is a Directed Acyclic Graph (DAG), a node may have several incoming or outgoing edges. These edges may only represent RaW dependencies, as false dependencies have been eliminated (WaR and WaW). Nodes without any incoming edges are known as *sources*, and nodes without any outgoing edges are known as *sinks*. The incoming edges are important to determine which task is going to be executed, as only source nodes can be considered as tasks candidates to be submitted in the Grid in a particular moment. The lack of incoming dependencies means that the task does not have to wait for any data generated in other tasks, so it can be the first to be executed. Besides, future generated tasks cannot create any incoming edge in any of the nodes which are already sources, as the new tasks go after them in the sequential order defined by the master program. Thus, all source tasks have their data dependencies solved, that is why we also call them *ready tasks*. The decision of what ready task must be executed between all of them is made in the resource broker, explained in Section 4.2.3.

When a task submitted to the Grid finishes its execution, our runtime receives a notification. In this case, the node that represents the task in the graph is erased, and with it all its outgoing edges pointing to tasks that depend on the one which has just finished. This is because data dependencies have been solved, as all result files in the task have been generated and are available for other tasks. Any of the nodes pointed by this task can become a source if it does not depend on any other tasks, thus it becomes a new candidate to be run in the Grid. This is the way the dependency graph is consumed by the runtime, when tasks finish their execution.

As the whole graph that represents an application is not needed in order to start scheduling tasks (whenever a source node is detected it can be submitted), the graph is created and destroyed at the same time dynamically while the application is being run. This decision also hides any possible overhead that the task graph generation may cause, as the generation time is hidden with the running time of the tasks. However, scheduling decisions are taken with parts of the graph, as the whole graph may have not been generated. Anyway, if the code has synchronization points (see Section 4.1.1) the task generation may be stopped in the master code, thus in some applications having the whole dependency graph generated before execution is not possible. Therefore, as far as the task is created in the Execute function, we try to schedule it if it does not have any data dependency and there are available resources in the Grid

We try to exploit the concurrency of the graph as much as possible, because many resources may be available in the Grid in order to submit tasks in a precise moment. If there exists more source nodes in the graph than free slots in the machines to run jobs, we will be able to schedule enough tasks to fill in all the available slots, thus we will be exploiting all the computing capacity of the Grid. However, the degree of parallelism of the graph depends on the programmed application: some parts may be completely parallel, without any dependencies between tasks, and others may be sequential due to data dependencies. In the second case, no parallelism may be applied.

4.2.3 Task scheduling: resource brokering

The Grid is an heterogeneous system formed by many different machines. If we want to submit jobs to the Grid, we need an entity which controls how many jobs are run at the same time in a machine, in order to administrate the resources properly. Without this control, the resources could become overloaded to a saturation point. The *resource broker* is the entity in charge of this.

In GRID superscalar we provide a simple resource broker which contains information about machines and provides free ones to run jobs. There are many research projects devoted to develop a resource broker or a meta-scheduler, where many different scheduling policies are studied. That is the reason why our objective is to provide a simple broker, not a complex one. Thus, in the future, our runtime could be adapted to work with more powerful brokers, such as the the GridBUS broker [107], the eNANOS broker [81] or GridWay [49].

Grid specification

In order to run an application GRID superscalar needs to know what is the Grid configuration: which machines can be used and their characteristics. This is provided with the *Grid configuration file*. Although the end user could provide this information, a more expert user such as a Grid administrator is the expected person to specify the Grid details. This configuration file can be generated by using the *deployment center* tool (see Section 3.1.2).

The information provided for every worker machine that belongs to the Grid is:

- Host name: fully qualified domain name of the host.
- Operating system: name of the operating system that runs on the host.
- Architecture: architecture of the host processors.
- GigaFlops: computing power of the host specified in GigaFlops.
- Network bandwidth: network bandwidth of the host in kilobits per second.
- Memory size: memory size of the host in Megabytes.
- CPU count: number of CPUs present in the host.
- SoftNameList: list of software available in the machine.
- Min port and Max port: port range to be used for machines that have inbound port restrictions.
- Limit of jobs: maximum number of jobs that can be concurrently sent to the machine.

- Queue: name of the queue to be used when submitting jobs. It may be specified to "none".

- Working directory: directory where the worker binary is located.

- Quota: disk space available in the working directory.

For the master machine, the host name, the working directory and the network bandwidth must be specified. The information provided is internally used in the runtime for different mechanisms that will be explained later: the network bandwidth to estimate the transfer time of a file between two machines (explained later in this section), the port range to send end task notifications with sockets (Section 4.2.4), and the quota to control the disk usage in the resource when using renamings in files (Section 4.2.4).

Some of these parameters are flexible in the sense that GigaFlops can be set as effective or peak, Memory as physical, free, virtual, ... The only requirement is that the same meaning must be used when users specify constraints for the jobs (see Section 3.2.5). The same happens with the parameter SoftNameList, to specify software available in the machine. The decision of how a software name must be specified is up to the user. All this data is stored in an XML file which is the Grid configuration file, and contains the initial configuration of the Grid for an application.

The Grid configuration file provides an initial set of machines to run the application, albeit it is very common in Grids that the pool of available machines changes dynamically over time: new machines can be used, or machines disappear from the pool. It may also happen that the configuration of a particular machine changes during the execution of a GRID superscalar application, because new software is installed, or because the machine can run more or less jobs than previously. In order to notify this changes to the runtime, GRID superscalar allows to read again the configuration file at execution time. This is done using a script named *read-configfile*, available with the distribution.

Therefore, whenever a change in the configuration must be done, the Grid configuration file can be edited, and the script read-configfile can be used to read again this information. The runtime will use the new Grid specified in the file (use new machines to submit jobs, machines removed from the file are not used, new limits of jobs are respected, and so on). The process of editing the file and notifying the runtime about reading it again is done by hand, and the information provided is static. However, an external resource broker with machine monitorization capabilities could periodically write this file and notify our runtime to read it again, making the information dynamic.

Scheduling policy

We have stated repeatedly that one of the objectives of the runtime is to reduce the execution time of the applications by using the Grid. With this objective, we must analyze the task run time in detail. The total time a task requires to be run in a remote resource is composed by two main factors: the time needed to transfer the task required input files from their original location to the resource, and the execution time the binary process is going to spend in the resource. If we want to reduce the total execution time of a task, we must reduce these two factors.

$$t = \mathit{FileTransferTime} + \mathit{ExecutionTime}$$

In our runtime we initially consider that the factor considering the binary execution time cannot be reduced. One of the problems is that the execution time of the worker depends on the computing power of the machine, and on the implementation provided by the user. In machines with several CPUs the process could be parallelized by the user with thread programming or OpenMP [71] to reduce the execution time, but at the cost of using more resources, which lets less room for other tasks to run. Using thread programming or OpenMP is opposite to our objective of keeping the Grid as transparent to the user as possible. Besides, in machines with a single CPU it could not be possible, or very difficult to reduce that time.

In contrast, the time to transfer input files is something that we can try to reduce from the runtime. That is why our policy to select the most suitable task for the available resources is focused in reducing this time. We will see that we exploit the *data locality* between machines to reduce even more this factor. The complete set of mechanisms that we provide to exploit this locality will be described in Section 4.2.4.

The runtime starts the interaction with the resource broker by asking for the set of machines available to run tasks. This is done in the runtime when a new task is created or when a running task finishes. The broker returns a set composed by all machines whose number of jobs submitted does not reach the specified limit of concurrent jobs. Then, for every task ready to be submitted, according to the data dependency graph, an estimation of the total execution time of the task is done (the file transfer time plus the execution time) for every machine available.

The file transfer time is estimated by defining the maximum transfer speed achievable between the source and destination machines (this is the minimum of both values, defined in the Grid configuration file). Thus, we can calculate the minimum time the transfer of files will last between machines. With this estimation, a less powerful machine with a gigabit ethernet connection speed available may be selected instead of a more powerful machine with a DSL connection speed, where the transfer time of files is bigger. If the input file can be obtained from different sources, the highest speed is selected first.

The execution time is estimated by calling to the cost function implemented by the user for that purpose (see Section 3.2.5). In that function, the time in seconds is calculated of what is the task supposed to last. The GigaFLOPS of the corresponding machine can be used, as well as the size of input files in order to calculate this time. We agree this is a very rough estimation of the execution time, however it is an initial solution. It is not our objective to develop a complex resource broker with performance prediction and study different scheduling policies. Besides, we have focused our efforts in reducing the transfer time in the formula, not the execution time. In the future more advanced predictors could be used, such as an external estimator from a resource broker.

Once the estimations have been calculated for all ready tasks in all available machines, the smallest of these estimations is selected as the task to be submitted to the Grid. Although this is a greedy policy, it pursues the idea of using the fastest resource with less file transfer time which is available at the moment of the task submit. This simple scheduling policy is enough to achieve the goal of reducing the total file transfer time in the application and exploit the locality of files. It will help to locate tasks that read/write the same (large) files in the same host, to reduce file transfers between the workers.

We believe our greedy scheduling is a good approach, as the Grid is a very dynamic environment that can ruin complex scheduling predictions very fast. Moreover, dynamic monitoring information of the resources (system or network load, memory usage, ...) should be used in order to try to implement a more advanced scheduling, instead of the static Grid configuration file that we provide. The brokering we do is static because of the static information of this file.

Match task constraints and machine capabilities

We have seen that GRID superscalar's user interface allows to define constraints for tasks defined in the IDL (see Section 3.2.5). These constraints follow the ClassAd syntax, and the ClassAd library is used to implement them [77].

The idea of ClassAds comes from the classified advertisements in the newspapers, where some people offers things, and other people demands things. This demand and offer can be matched. A ClassAd is a mapping from attribute names to expressions. In the simplest cases, the expressions are simple constants (integer, floating point, or string) but they can be more complicated. A ClassAd is thus a form of property list (a list of key - value pairs). The ClassAd library allows to build ClassAds and match them to determine their compatibility.

In GRID superscalar, when GS_On is called, ClassAds are generated for every machine defined in the Grid configuration file. The ClassAds contain as attributes the ones defined in the configuration file (architecture, operating system, ...), and this is what machines offer to jobs. Machines do not have any requirement for the jobs they run.

Tasks in GRIDSs do not offer capabilities, but demand them. So, the ClassAd to be built for a task does not contain any attribute. Only the special attribute *Requirements* is specified, which will be used for the matching process. The value of the Requirements attribute is defined by the user with the constraints function provided by GRIDSs. In order to generate a ClassAd for a task, the corresponding constraints function is invoked by the runtime.

When ready tasks must be scheduled in the runtime, first ClassAds are built for all of them. Before estimating the execution time of a task in an available machine provided by the broker, the task ClassAd is matched with the machine ClassAd. The estimation is only performed if the requirements are accomplished. This matching is done by evaluating the *Requirements* attribute against each other ClassAd. As machines does not have any requirements, their attribute has always value *true*. Task requirements are evaluated against machine attributes, and only if the evaluation returns *true* the requirements are passed.

If current task does not match with any available machine, the runtime must ensure that there is a machine able to pass the requirements of this task. So, the ClassAd is matched against all machine ClassAds (available or not) to ensure that the task can be executed in the future when the machine or machines that pass its requirements are available to execute jobs. If no machine matches the requirements, the execution is stopped.

ClassAds include an extra parameter used to implement a mechanism to control the available quota in workers. Thus, the quota is considered a requirement to be fulfilled also by machines. The quota control mechanism will be detailed in Section 4.2.4

4.2.4 Task scheduling: file locality exploitation

In previous section we have argued why we try to reduce the time to transfer input files to resources. With this objective, we implement the exploitation of file locality in the Grid when running a GRID superscalar application. The larger is the data required by an application to be run, the better will be the results obtained exploiting the file locality. As our target applications are those whose main data parameters are files, the size of these files can be very big. Moreover, moving the data around all the time is not efficient in terms of network usage.

Keep remote copies of files

In order to run a job from a source to a destination machine in the Grid, some systems choose to transfer input files to a temporal directory created in the destination machine, execute the job leaving the results in the directory, transfer the results back to the source machine, and erase everything in the destination machine as if no job had been executed. An example of a system using this approach is GridWay [49]. This may be good for an environment where jobs are considered as independent entities, but in our case, jobs submitted from GRID superscalar have the relationship that they belong to the same application, and they may reuse input data or results from previous jobs.

In our runtime, whenever a job is submitted to the Grid, input files are transferred first to the worker machine, but when the execution of the worker finishes, they are not erased but kept there. The same happens with result files of a task. Our runtime keeps track of the location of every file distributed or generated in a worker, so whenever an estimation of the total execution time of a task is done, those files that are not required to be transferred are not added to the estimation. It may even happen that no file transfers are required to run a task in a particular machine. The scheduling policy proposed in previous section together with this locality mechanism is able to reduce dramatically the file transfers.

Besides, different job submissions to a single machine which share input files are coordinated by the runtime. The first submission done to a machine starts the needed file transfers, and if there is a new submitted task which shares any input file with the first, this new task will not transfer the common files again but wait for the file transfers of the first task to finish.

The input files of a task are transferred to the working directory of the machine, and softlinks point to them in the temporal directory created to execute the task. Output files are generated in the temporal directory, but moved to the working directory when the execution of the task is over. Temporary files created during the task's execution are erased when the temporal directory is destroyed. Only the files defined as inputs or outputs of the task will remain in the working directory to be reused.

The case of output files is specially important when saving transfer time. It is very inefficient that, in order to use a result file, we must wait that it is transferred back to the master machine when the task has finished, and transferred again to a new destination. In GRID superscalar, when the task finishes the result can be directly used by a new task in the same machine, or transferred to a new machine (in the worst case, one transfer, in contrast to the two transfers needed in the other scenario).

In summary, we try to allocate tasks to resources where input data is already available to save those file transfers. However, if the machine that requires less file transfers is busy executing other tasks, it will not be used. The required files will be copied to a new machine, having several copies of the same file in different machines. Our policy is to use the best resource available at the scheduling moment of the task (where the best means the one with less estimated total execution time). In general, files are transferred on demand when a task needs them, if the transfer cannot be avoided.

Having several copies of the same file distributed around different machines makes mandatory the implementation of a coherency protocol. When a task writes in a single file which is available in several machines, the rest of the copies of the file are not valid anymore and must be marked as invalid. Invalid copies are deleted during the execution of the application. We follow a relaxed consistency model, as we do not update all copies of a file when one of them is written because this could increase the network traffic.

The GRIDSs runtime implements a hash where several information about files is maintained, in order to avoid querying a remote machine to find out the information. For every file involved in the computation, its name and size is maintained, together with some flags to mark if the file is valid in the master machine, or if it is pending to be deleted, and the disk names where the file is available. For each disk, several flags are maintained, such as if the file is valid, if it must be get to the master because it is a result, if it is currently being transferred to the disk, and so on.

There already exist tools to have replicas of files distributed in the Grid. Even the Globus Toolkit includes one: the Replica Location Service (RLS) [23]. Nevertheless, we have implemented our own replica system for two main reasons. The first one is that we need a very specific functionality in order to implement the coherency protocol, which is not provided by RLS. Replica systems seem to be more focused to provide a single logical name to different physical names of files, and they do not include features to implement coherency protocols. The second reason is that we do not depend on any replica service if we want to port our runtime to a different Grid middleware. This has been proven to be very useful when porting the GRIDSs runtime to the ssh/scp tools [9].

The main drawback of exploiting the locality is that disk usage is increased in the Grid, more precisely in the workers, as copies of a single file may be placed in several machines. This becomes worse with the renaming techniques introduced in the runtime, because a single file may have several versions of itself, and they may be distributed between machines as well. Nonetheless, our objective is to reduce the execution time of the application, so the benefits for us are higher than the cost paid. Moreover, mechanisms to control the quota in the workers and to erase unused versions of files have been introduced to reduce the disk usage (explained later in this section).

Shared input disks and working directories

To keep exploiting locality between machines in our Grid, two more mechanisms have been added related with sharing disks between machines. The first one is the ability of the runtime of working with input disks which are shared between machines (we call them *shared input disks*), and the second one is to share a common working directory between some of the workers (we call them *shared working directories*), which is specially useful if in the Grid there are one or more clusters defined.

A machine usually has access to its own disk, but some other disks may be mounted using shared file systems such as NFS [84] in order to share data between machines. This may be used to access large input files, large databases, or similar. GRID superscalar includes the possibility of declaring input disks which are visible for several machines in the Grid. In order to estimate the file transfer time of a task, whenever an input file passed to a task is located in a shared disk, the runtime checks if the candidate machine is able to directly see the shared disk. If this is true, no transfer will be required between the master and the worker if the task is executed in this resource, because the file can be referenced directly from the worker, thus the transfer time is not added in the estimation.

These shared input disks are declared giving a virtual name to the shared input disk, which corresponds to a root path where the disk is mounted in the master machine. In the worker machines which can see this disk, the same virtual name is given to the disk, but the root path where this disk is mounted in the worker machine may be different, thus it is also specified. In practice, what the runtime does is to translate any reference made to a shared input disk in the master, to the corresponding path in the worker, so the worker process can access it directly. As the only translation in the file reference is the root path, subdirectories can be freely used in this shared disks, as they will be kept in the translation to the new path.

Declaring replicated areas in different disks as shared input disks is also possible. If two paths in two different machines have the same data replicated, there is no need to transfer files from one to another, as they both can use their local copy to perform a more efficient access. Therefore, for GRID superscalar replicated areas behave exactly as input disks which are really shared between machines: file transfers can be avoided. The existence of a real shared disk is transparent to the runtime. Replicating data between machines in advance is very useful if several machines work with the same input database, and specially if the database is very big. A local access is more efficient that waiting for a data transfer from a remote machine.

Files in a shared input disk can only be read. We do not allow to write in a shared input disk. This is mainly because we do not distinguish between a real shared disk or a replicated disk when referenced by our runtime. Writing in a shared disk is possible if the disk is shared in reality. However, if the disk is a data replica, a write in the disk will not be propagated to the rest of replicas, leaving data not consistent between them. Besides, even if the disk is a real disk mounted for several machines, we do not know which is the exact physical location of the disk (i.e. if the disk has a fast local access in a machine, or an access through a shared file system must be performed, which means in the end transferring the file). Thus, our policy for writing results of a task is always to write them in the working directory where the worker has been run.

In an heterogeneous Grid, many different types of machines may be used, from single PCs to clusters. A cluster may be declared as a single machine, if it has a local resource manager, such as a queuing system. However, if no local resource manager is available and nodes can be accessed directly as if they were independent machines, it can be described to GRIDSs that way. Clusters commonly have a shared file system mounted, which is used to share data between all the machines in the cluster. In GRID superscalar, we allow the possibility of declaring *shared working directories* between machines.

A working directory in GRIDSs is the location where the worker is executed in a machine, where input files are transferred and output files are generated for a task, and both are kept there to exploit the locality of data. Therefore, if two or more machines share their working directory, all these files may be accessed directly and no remote transfers will be needed between them. When an input file is transferred to the disk, all the machines will be able to access it. The same happens when a task finishes and writes its output files: the files are written to the disk, and are immediately available for the rest of the machines which share the working directory.

As explained with shared input disks, shared working directories are also declared using a virtual name for the disk and the path used to access it from the corresponding machine. So, several machines may use the same virtual name for their working directory when the working directory is shared between them. Sharing the working directory is not only possible between workers but also between the master and one or more workers. As the disk may be mounted using different paths in the machines, internally GRID superscalar translates the specified paths from the original to the new one, as it is done with shared input disks.

It is also possible to declare two machines that have their working directory in the same physical disk, but mounted in different directories. In this case, the runtime will treat them as separated areas, and file transfers will be requested from one working directory to another. When a file transfer is specified taking as a source and destination the same machine, in practice the Grid middleware service translates the transfer to a local copy.

When declaring this situation, different virtual names should be used for the directories. Anyway, we do not recommend to use this configuration because the optimal solution in this case is to declare a common working directory for the two machines.

Erasing unused verions of files

We have previously seen how the renaming techniques used in the runtime are able to improve the intrinsic parallelism of the application by eliminating data dependencies between tasks (see Section 4.1.2). When a new renaming is generated for a file, a new version of the file is created which exists together with the original one. As renaming may be applied many times for the same file, many different versions of the same file may exist at the same time. This may be a problem in terms of disk usage if we allow all versions to exist during the whole execution of the application. The lifetime of a file version must be studied in order to determine for how long the version will be needed in the system. Whenever the version is not needed anymore, we will be able to delete it, therefore reducing the disk usage caused by renaming.

We know in advance that new calls in the code using a file which has been renamed must be translated by the runtime always to the last name of the file, because of the sequential order defined by the master main program. The newest version of a file cannot be erased in any case, as new tasks may read it, but old versions are candidates to be erased. This means that a file version lifetime is determined by the tasks that need to read that version, and by the existence of a newer version. Therefore, once nobody wants to read a file version, and a new renaming exists, the file can be eliminated from the whole system (from the runtime and from all the working directories in the machines), as a new version exists and no task is going to read that old version.

The runtime controls during all the execution of the application the reading accesses to files. When the task is created, input files determine that a new reader requires those files, so the number of readers for each file is increased. When a task finishes its execution, input files are checked in order to decrease the number of readers for every file. If any of these files reaches the number of zero readers, and it is not the latest version of a file, it can be deleted.

Figure 4.10 shows a source code which causes several renamings at the same file. More precisely, a new version of f1 is created in every iteration to avoid WaW and WaR dependencies with next iterations. If we name f1.X to the renamed files, where X stands for the iteration where the file is generated, we may have ten versions of f1 active at the same time (f1, f1.1, f1.2, ..., f1.9) because in every iteration there exists a task T2 which is going to read that version. Once T2 is executed, the corresponding f1 will be erased as

```
for(i = 0; i < 10; i++)
{
  T1(..., ..., "f1");
  T2("f1", ..., ...);
}
```

Figure 4.10 Code which causes file renaming

no more readers need that file and there exists a new version. The file f1.9 is the latest version of f1, thus it will not be erased even if currently no readers are waiting to read its value.

We can determine what would be the worst and the best case of disk usage caused by renaming. In the best case, only the latest version of a file is maintained, as older versions have already been deleted because no task is going to read them. In the worst case, all versions may be maintained, because all tasks that must read those versions have not been executed yet. However, as our goal is to reduce the execution time of the application, we assume this temporary higher disk usage to achieve that objective, knowing that, as soon as the version is no more needed it will be erased.

Disk quota control

Previous sections have presented how the locality mechanism implemented in GRIDSs helps to reduce the file transfers between machines, at the cost of using more disk space in the machines which form the Grid. Also, the renaming mechanism used to increase the parallelism of the application increases the disk usage. Disks have a limited size, and users commonly have a maximum space assigned to be used, also known as *quota*. Users may be aware of their quota in a machine, in order to know if there is enough quota to execute a task. However, because of the GRIDSs mechanisms, the quota available in the machine may be less than initially expected.

Also because of the parallelism exploitation, several tasks may be running at the same resource in a given moment. If these tasks have common source files, the quota used at a precise moment will be only a single instance of the files, not an instance for every task. A user may be confused thinking that only an instance of a task can be executed, while it is not the case thanks to the locality exploitation. In this case the disk usage would be reduced when running several tasks in the same resource. A mechanism is needed in order to control the quota usage of GRIDSs in the worker machines to determine if a task is able to be executed due to its quota requirements.

The GRID superscalar runtime controls the quota used in every machine specified as a worker. Whenever input files are transferred to a worker, or result files are left in a worker, the quota available in that machine is decreased. Also if files are deleted due to invalidation of copies between machines or because they are unused versions of files, the quota is increased. The quota is defined as an extra attribute of the machine which may be specified in the Grid configuration file which describes the machines used as the Grid to run an application.

Before executing a task, the runtime calculates the required disk space to transfer the input files to the candidate machine. Therefore, the quota requirement for input files is automatically controlled. Unfortunately, some quota problems may still arise, as the runtime is not able to determine in advance the size of the output files or the temporary files created by the user in the task. Anyway, if the user is aware that result files may be a problem in terms of quota, the constraints mechanism allows to specify as a requirement of a task a minimum quota available in the machine where it must be run. This quota specified by the user (when available) is added by the one determined by the runtime to know the total quota requirement of the job.

The quota is defined as requirement of a task, so it is also used in order to filter resources. It is added to the task ClassAd as a requirement and matched with machine ClassAds in order to determine if the machine is able to accomplish the requirements of the job, as described in Section 4.2.3. So, some machines may be discarded to be used when they are available if there is not enough quota to run a particular task.

4.2.5 Return of task results and end of application

When the execution of the application finishes, there is a post-process executed in the GS_Off call where all working directories in the remote hosts are cleaned, application output files are collected and copied to their final locations in the master machine (as they have not been collected due to locality exploitation) and all is left as if the original application had been run locally, following our objective of keeping the Grid as invisible as possible to the user. Thus, for the end user the application starts and finishes as any other local application, with the only difference that hopefully the execution has been much faster by using the Grid resources.

Due to the file locality implemented, task file results are not returned to the master, but left at workers. However, other information must be provided to the runtime when a task finishes, such as the exit status of the task, information about the size of result files kept in the worker and output scalars generated by the task. This information is sent in a special message from the worker to the master, independently from any messages that the Grid middleware may send to the GRIDSs runtime. We have implemented two mechanisms to send this message: using a result file, or using a direct socket connection. We will discuss later in this section the advantages and drawbacks of both.

Task output scalars

Our runtime takes into account data dependencies based on files and not between scalars. The discussion of this decision has been already described in Section 4.1.1. Whenever a task returns an output scalar, other tasks or inline code in the user main program may need it immediately. Reading this parameter without any synchronization could lead to an erroneous execution, as it will not be correct until the task that generates it finishes. The only way to guarantee the correct execution of the application is to execute a *local barrier synchronization* when a task has one or more output parameters which are not files. By local barrier synchronization is meant that the runtime will temporally suspend the execution of the user main program at that point until the task generating those outputs has been executed. The wait time in the barrier could be hidden in the execution time if GRID superscalar has enough tasks available to be run in parallel, and the task with the output scalar is early scheduled for execution. If the application do not meet this conditions, performance may diminish due to stopping the generation of tasks in the master code.

Although this behavior may seem inefficient, it is reasonable under the assumption that tasks generally will have files as output parameters and exceptionally some tasks can have a scalar output parameter. We differentiate that local barrier from a global barrier, which will wait until all submitted tasks end. Besides, the output scalar could be written to a file and passed as a result file to avoid this local barrier if needed.

End of task messages using files or sockets

The runtime supports two ways of sending notification messages from the worker when a task has finished: by sockets or by files. The socket mechanism is handled by a thread created in the master program which listens for notifications from workers and receives their content. The notification message contains the output scalars of the task, together with an error code from the task specified by the gs_result parameter useful to detect task failures. In addition, the size of the output files generated by the task is also sent to the master, so the runtime can gather this information to perform future scheduling decisions.

The file mechanism consists on writing the same message used in the socket mechanism to a file. This file is generated by the worker when its execution finishes and is specified when the job is submitted to be transferred back to the master when the task finishes. Thus, the master can open this file to read the content of the message.

If we compare both mechanisms we can see that the socket mechanism allows a faster communication between the worker and the master, because the message is directly received, while in the file mechanism the Grid middleware must transfer the file back to the master so the runtime can open it to read the message. However, the requirement of the socket mechanism is to be able to open a direct connection from the worker machine to the master, and this cannot always be accomplished by worker machines. An example of this limited connectivity is commonly found in clusters, where only a front end node can establish direct connections to external machines, and the rest of nodes are only allowed to communicate between them. Thus, when clusters or machines with limited connectivity are used in our Grid, only the file mechanism could be used.

The socket mechanism also allows to skip any possible overhead wich may be introduced by the Grid middleware when notifying the end of a task. An example of this overhead can be found in the Globus Toolkit version 2 job execution service. In order to detect the end of a task, a job manager process polls every 30 seconds to check if the job has already finished, as it is demonstrated in Section 4.3.1. In the worst case, almost 30 seconds of overhead may be introduced in a job execution, and if the application is formed by thousands of tasks, the overhead may be important. With the socket mechanism, whenever the message is received the task is considered to be finished, without having to wait for the end notification from the Grid middleware. Therefore, we recommend to always use this mechanism when workers in the Grid do not have limited connectivity to reduce the execution time of the application as much as possible.

4.3 Evaluation tests

The main objective of the runtime is to achieve a faster execution of the application by using the Grid resources. We have evaluated the execution of different applications to see how we achieve this objective. Our first goal is to make a deep analysis of the runtime, in order to check that we can speed up applications and to analyze possible bottlenecks in our system.

The second thing we want to demonstrate is that we are able to obtain speed up from the executed applications, but this speed up depends on several factors of the application: the granularity of tasks and the parallelism available in the application. We have selected a simple example and several real use cases of GRID superscalar to study the scalability of applications and compare the performance when using our environment.

Table 4.1 Summary of presented applications

Application	Interesting features
NAS Grid Benchmarks	Representative benchmark, includes different types of workflows which emulate a wide range of Grid applications
Simple optimization example	Representative of optimization algorithms, workflow with two-level synchronization
New product and process development	Production application, workflow with parallel chains of computation
Potential energy hypersurface for acetone	Massively parallel, long running application
Protein comparison	Production application, big computational challenge, massively parallel, high number of tasks
fastDNAml	Well-known application in the context of MPI for Grids, workflow with synchronization steps

Table 4.1 contains a summary of the applications we present in next subsections. We have also detailed the most interesting features of each application.

4.3.1 In depth analysis

In this section we present some of the results obtained with the implementation of the NAS Grid Benchmarks [11]. We analyze the scalability of the benchmarks when run in our testbed. Besides, we have instrumented the runtime to obtain Paraver tracefiles [59] in order to do a performance analysis, which allows us to know precisely the behavior of the runtime and detect possible bottlenecks to be improved.

The NAS Grid Benchmarks (NGB) [28] are based on the NAS Parallel Benchmarks (NPB). Each NGB is a Data Flow Graph (DFG) where each node is a slightly modified NPB instance (BT, SP, LU, MG or FT), each defined on a rectilinear discretization mesh. Like NPB, a NGB data flow graph is available for different problems sizes, called classes. Even within the same class there are different mesh sizes for each NPB solver involved in the DFG. In order to use the output of one NPB solver as input for another, a interpolation

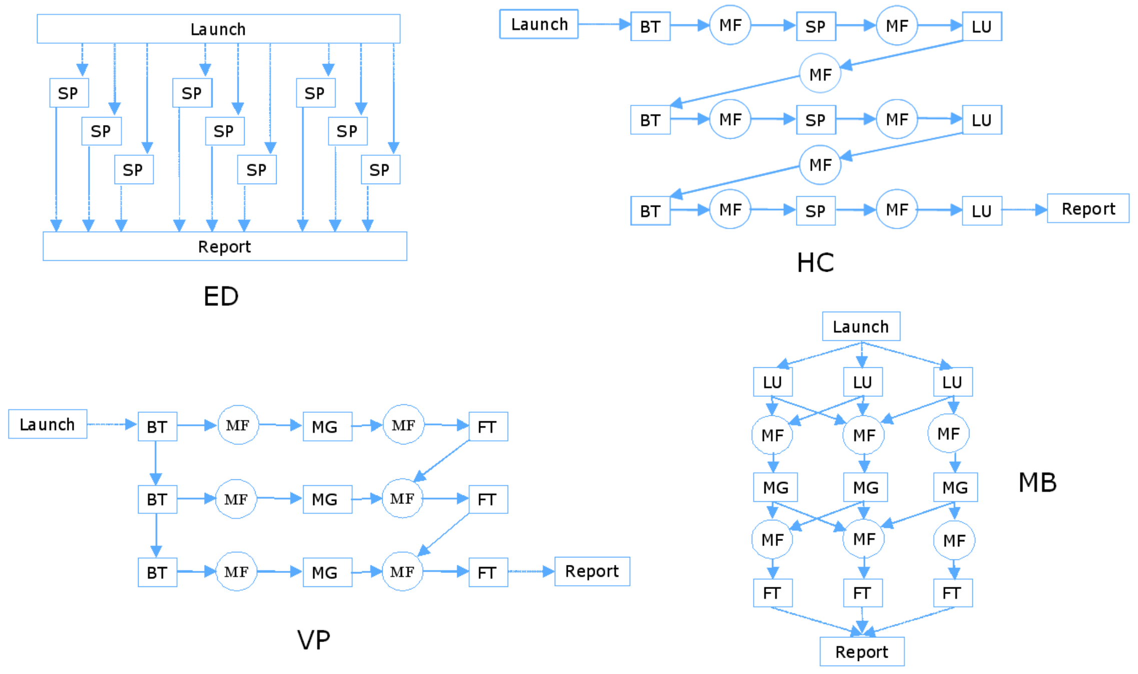


Figure 4.11 Data Flow Graphs of the NAS Grid Benchmarks

filter is required. This filter is called MF. Four DFG are defined, named Embarrassingly Distributed (ED), Helical Chain (HC), Visualization Pipe (VP) and Mixed Bag (MB). Each one of these DFG represents an important class of Grid applications.

- ED represent a parameter study, which is formed by a set of independent runs of the same program, with different input parameters. In this case there are not data dependencies between NPB solvers.

- HC represents a repeating process, such as a set of flow computations that are executed one after another. In this case a NPB solver cannot start before the previous one ends.

- VP represents a mix of flow simulation, data post-processing and data visualization. There are dependencies between successive iterations of the flow solver and the visualization module. Moreover, there is a dependency between solver, post-processor and visualization module in the same iteration. BT acts as flow solver, MG as post-processor and FT as visualization module.

- MB is similar to VP, but introducing asymmetry in the data dependencies.

Figure 4.11 shows the DFG of the four benchmarks for classes S, W and A. If we analyze the DFGs we can see that he maximum parallelism of ED is 9 for all of these classes. HC is totally sequential and for MB and VP the maximum task parallelism is 3. A paper and pencil specification is provided for each benchmark. The specification is based on a script file that executes the DFG in sequence and in the local host. For each benchmark a verification mechanism of the final data is provided.

We have implemented the scripts that execute the DFG using the GRID superscalar programming model. However, a modification of the NPB instances was needed to allow GRID superscalar to exploit all its functionalities. We modified the code of the NPB instances in such a way that the names of the input/output files are passed as input parameters. In the original code each NPB instance generates internally these names. The file names were generated in such a way that they were different in each execution of the same NPB program. With our modification we can reuse the same file name in different iterations, and GRID superscalar guarantees the proper execution using the renaming feature. In this way, the NGB main program is much simpler than the original one.

We ran all the benchmarks in a local testbed formed by: Khafre, an IBM xSeries 250 with 4 Intel Pentium III, and Kadesh8, a p630 node of an IBM RS-6000, formed by 8 SP nodes with 16 Power3 processors each, and 9 p630 nodes with 4 Power4 processors each. Khafre and Kadesh8 ran tasks as workers, and a PC based system with Linux was used as the master. In each case, a maximum number of tasks that could be sent to each machine was set, which can vary from 1 to 4 (as both machines have 4 CPUs in a single node).

Tables 4.2, 4.3 and 4.4 show the results for the VP and MB benchmarks when run with classes S and W. We have not detailed executions with ED and HC since their DFGs are less interesting (completely parallel and completely sequential respectively). The benchmarks were run assigning from 1 to 4 tasks to each worker. The times reported in the tables are average times from several executions, since the total execution time can vary more than 10% from one execution to another.

Figures 4.12, 4.13 and 4.14 show the relative speed up achieved in the executions. MB scales with the number of tasks as expected in both machines used alone, but up to three

Table 4.2 Execution times for the NAS Grid Benchmarks MB and VP on Kadesh8

Benchmark	1 task	2 tasks	3 tasks	4 tasks
MB.S	294.14 s	166.57 s	152.05 s	154.15 s
MB.W	543.31 s	298.14 s	223.60 s	225.15 s
VP.S	310.16 s	280.52 s	252.89 s	248.05 s
VP.W	529.65 s	336.96 s	339.55 s	339.08 s

Table 4.3 Execution times for the NAS Grid Benchmarks MB and VP on Khafre

Benchmark	1 task	2 tasks	3 tasks	4 tasks
MB.S	243.70 s	133.25 s	96.46 s	110.69 s
MB.W	591.88 s	321.55 s	222.18 s	225.98 s
VP.S	330.56 s	248.07 s	252.77 s	251.87 s
VP.W	518.40 s	310.39 s	320.21 s	320.32 s
VP.A	1663.21 s	1243.04 s	1267.77 s	1128.61 s

Table 4.4 Execution times for the NAS Grid Benchmarks. Kadesh8 and Khafre simultaneously

Benchmark	1 + 1 task	1 + 2 tasks	2 + 1 tasks
MB.S	278.55 s	172.85 s	209.92 s
MB.W	348.72 s	314.83 s	365.99 s
VP.S	317.88 s	314.54 s	314.96 s
VP.W	373.85 s	249.11 s	232.14 s

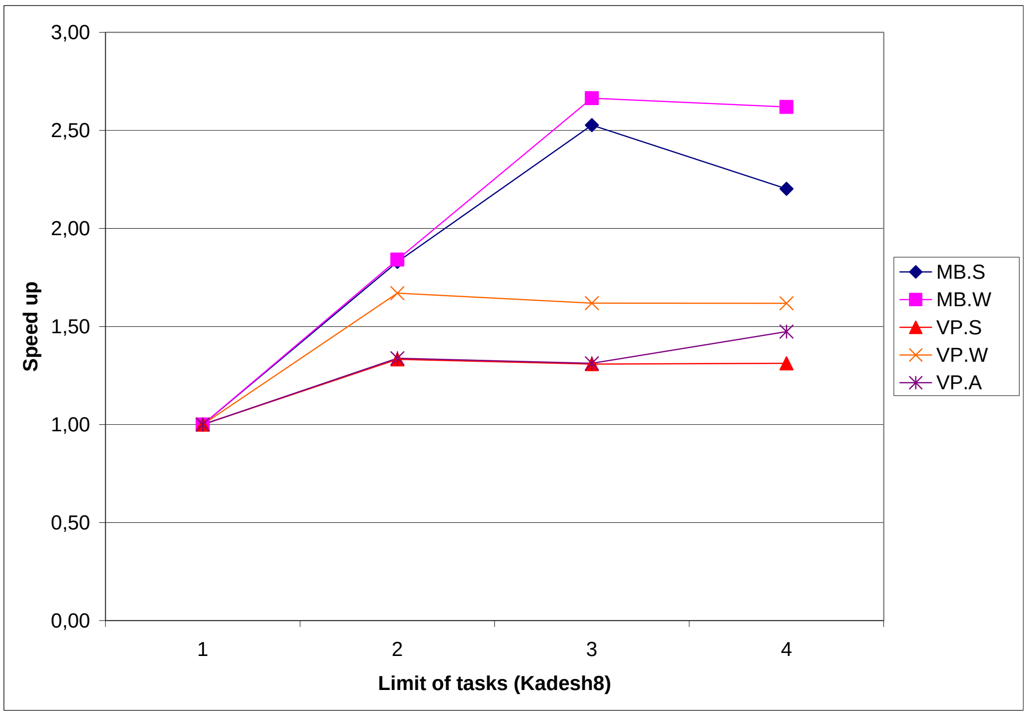


Figure 4.12 Speed up using Kadesh8

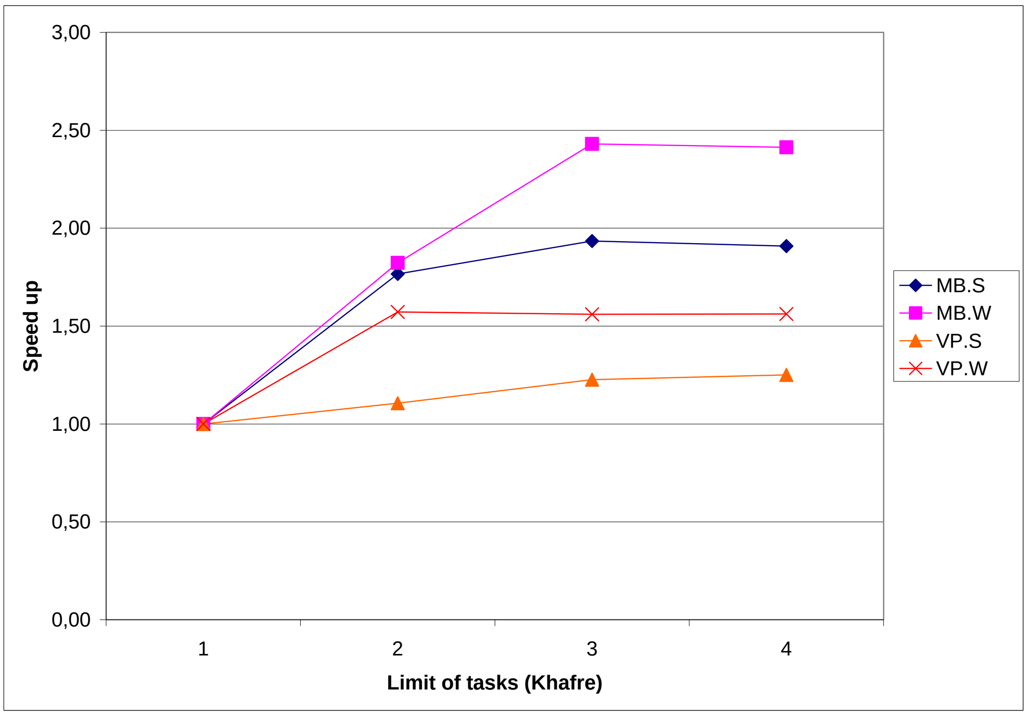


Figure 4.13 Speed up using Khafre

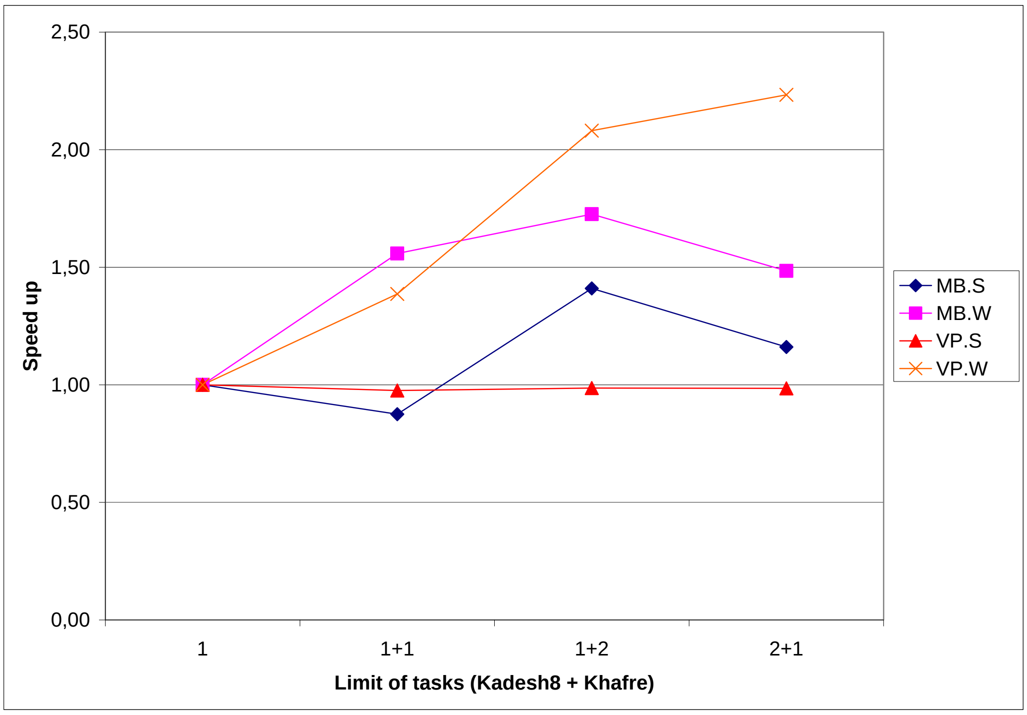


Figure 4.14 Speed up using Kadesh8 and Khafre

workers, because the maximum parallelism in the DFG is 3. VP is not scaling nicely with the number of tasks. As we want to perform a detailed analysis of our runtime, the VP benchmark is a good candidate to do a performance analysis, which is reported later in this section to understand why it does not scale. When both workers are used the time is not scaling as expected. The reason for that behavior is also analyzed later.

In order to do a performance analysis of the benchmarks the GRID superscalar runtime was instrumented to generate Paraver tracefiles. Paraver [59] is a performance analysis and visualization tool which has been developed at CEPBA for more than 10 years. It is a very flexible tool that can be used to analyze a wide variety of applications from traditional parallel applications (MPI, OpenMP or mixed) to web applications.

The traces generated for the GRID superscalar applications were only in the master side. However, to take into account the overhead of Globus, time measures of the duration of the workers tasks were also performed. To generate the traces for the GRID superscalar applications, two kind of elements (which are the base of the Paraver tracefiles) were used: the state and the events. The state of the GRID superscalar can be, for example, *user* (when running user code), *Execute* (when running runtime primitives), and so on. Additionally, events were inserted to indicate different situations: beginning/end of callback function, task state change (job request, active, done, ...), file open/close, etc.

Some of the results for the NGB benchmarks presented in Tables 4.2, 4.3 and 4.4 seem to be unreasonable at a first glance. For example, VP benchmark, class W, when run in the IBM Power4 node. As the maximum parallelism of this benchmark is 3, it is not surprising that no benefit is obtained when using the 4 processors (4 tasks). However, we expected to get better performance with 3 tasks than with 2, therefore this two cases were re-run and Paraver tracefiles were obtained.

Table 4.5 shows the time the application is in each state for different runs. It is observed that the main part of the execution of the application is in the Execute function. Analyzing with more detail it can be observed that there are 16 different Execute bursts in the tracefiles, one for each of the tasks of the graph (see Figure 4.11) plus one for a GS_FOpen performed at the end of the benchmark.

Table 4.5 General performance information for the NAS Grid Benchmark VP.W

Task #	User	Execute	GS_On/GS_Off	Barrier	TOTAL
4 tasks	0.002 s	325.579 s	13.494 s	11 s	339.08 s
3 tasks	0.002 s	347.159 s	23.652 s	12 s	370.81 s
2 tasks	0.002 s	318.360 s	28.583 s	11 s	341.946 s

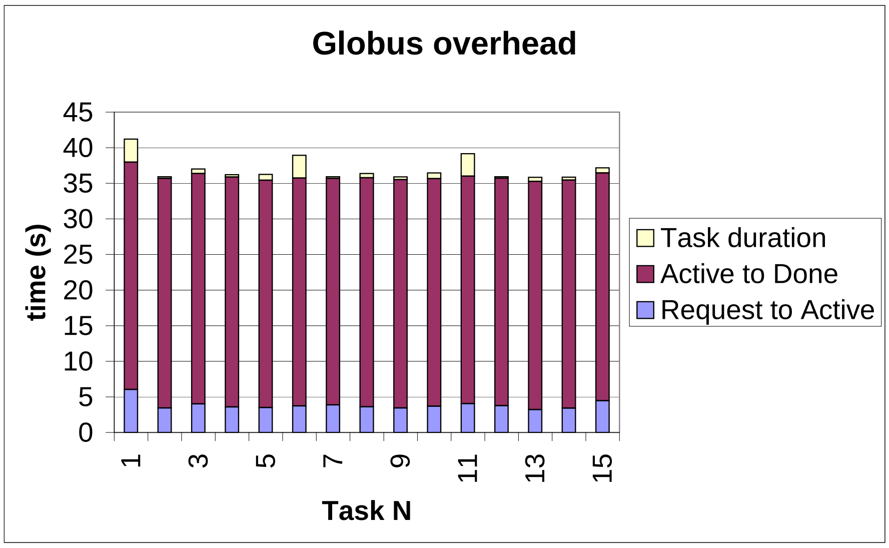


Figure 4.15 Task elapsed time composition for the NGB VP.W

Next step was to identify for each task, the time invested in each step. In Figure 4.15 we can see for each task the elapsed time for different steps of the task (except the last GS_FOpen task, which is locally executed and therefore no Globus events are inserted in the tracefile). The figure shows for each task:

- Request to Active: the time elapsed between the client does the job request until the callback function enters notifying that the job is in the active state.
- Active to Done: the time elapsed between the callback notifying that the job is in the active state until the callback notifying that the job has ended.
- Task duration: elapsed time of the task measured in the worker side (this time is included in the Active to Done time, but it is indicated separately to outline the difference).

It is observed that for each task, the Request to Active time is in average 3.86 seconds and the Active to Done 32.06 seconds. However, the average elapsed time of the tasks in the worker is 1.03 seconds. The Active to Done has an average value rounding the 30 seconds. This time matches the GRAM Job Manager polling interval. This has been reported before in other works [96]. This polling interval can be changed editing the

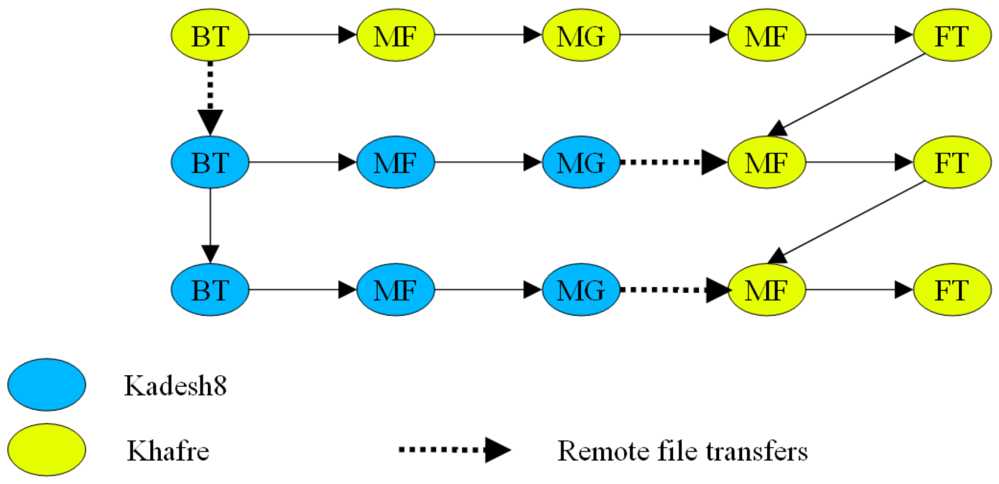


Figure 4.16 NGB VP: task assignment to workers

GRAM sources. However, if the granularity of the tasks is large enough, this polling interval would be reasonable.

Regarding the performance between the 2 and 3 tasks cases, the corresponding schedules between both cases were observed with some detail. Although the VP data flow graph has a maximum parallelism of 3, this maximum parallelism is only achievable for a part of the graph. With a correct schedule, the same performance can be achieved with 2 tasks as with 3 tasks. The reason why GRID superscalar is not scaling when using 3 tasks is because the schedule with 2 tasks is good enough to get the maximum performance.

We describe now the results of the analysis of the NAS VP, class S, when run with two workers. The results shown in Table 4.4 when one task is assigned to each worker are worst than the case when one task is assigned to worker Khafre (Table 4.3. In this case, we are directly analyzing the task schedule. Figure 4.16 shows the assignment of each VP task to each worker. The tasks in light color have been assigned to Khafre, the faster worker, and tasks in dark color have been assigned to Kadesh. The dashed line between tasks assigned to different workers represent that a file transfer is required between both workers. Regarding the assignment, we consider that it is correct, since GRID superscalar has assigned more tasks (and the tasks in the critical path) to the faster worker and also the number of file transfers between both workers is very low (only 3 file transfers).

We can compare the transfers done with GRIDSs exploiting the locality with an approach that does not maintain any locality to see the potential gain this mechanism is able to provide. There are two file transfers that both approaches must do, the initial (to provide the first input file) and the final (to collect the last result file). The rest of the transfers can be calculated with the graph shown in Figure 4.16. Every edge in the middle

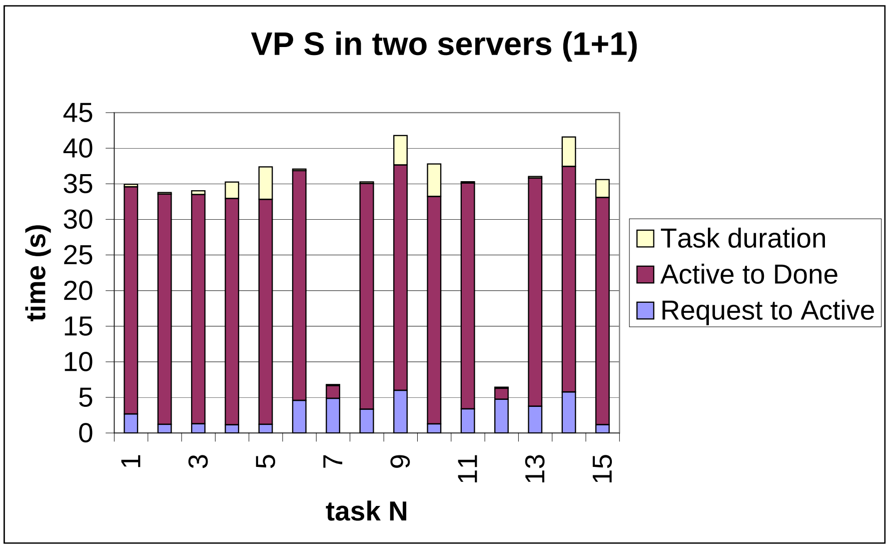


Figure 4.17 Task elapsed time composition for the NGB VP.S

of the graph means 2 file transfers when no locality is considered (one from the worker to the master, and another one from the master to the machine which needs the data). In contrast, when file locality is considered, solid edges mean no transfers are needed, and dashed ones correspond to one transfer (from a worker to another worker). This means that file transfers with locality have been reduced to a 14.7 % of the transfers that would have been done without considering any locality. In this case, as files are small in size and the network connection is very fast, the time gained is not very significative. However, the bigger the files and the slower the network that connects the machines in the Grid, the better will be the gain obtained with this mechanism.

Figure 4.17 shows the elapsed time for each task in the two different workers. In this case, the Request to Active time is different if the task is assigned to one worker or the other. In Khafre the average is 2.4 seconds and in Kadesh8 is 4.1 seconds. The overhead of the transfer time from Kadesh8 to Khafre in tasks 9 and 14 makes that the Request to the Active time in these two tasks is above the average in Khafre (6 seconds and 5.8 seconds respectively). Also, for task 6, which receives a file from task 0, the Request to Active time is above the Kadesh8 average (4.6 seconds). The Active to Done time is again around 30 seconds for almost all cases except for two tasks, for which it is around 1.5 seconds. In those two cases, the poll entered much earlier than in the other cases and the end of task was detected with a much shorter time.

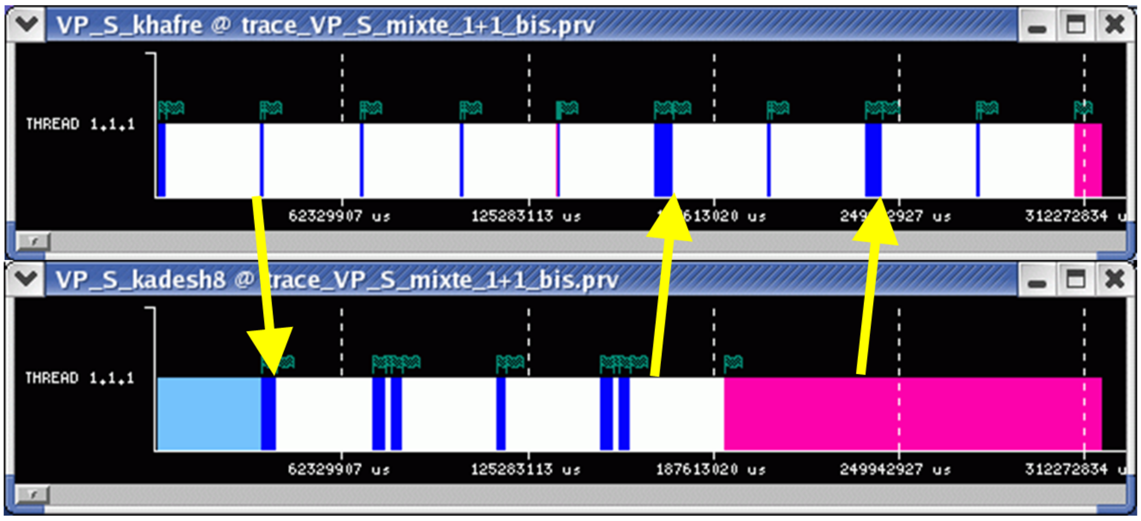


Figure 4.18 NGB VP.S: task scheduling when two workers are used

Finally, Figure 4.18 show the task schedule for the tasks in each worker. Those plots are Paraver windows. The window in the top shows the tasks assigned to Khafre and the one in the bottom the tasks assigned to Kadesh8. The segments in dark color represent the Request to Active state. During this time the file transfers (if necessary) are performed. The segments in white represent the Active to Done state. The file transfers between both workers have been highlighted with strong light color lines. Since the overhead of the file transfers is no more than 15 seconds and the schedule is appropriate, it is difficult to understand the low performance achieved with this benchmark when using two workers. The reason is detected comparing this execution with the one worker version: to allow the correct execution between the two heterogeneous workers the benchmark is run in ASCII mode. For example, VP.S when run alone in Khafre with two workers in ASCII mode takes 312.14 seconds, which is above the 280.52 seconds when run in binary mode. Also, MB.S when run alone in Khafre with two workers in ASCII mode takes 223.46 seconds, which is again above the 166.57 seconds obtained in the binary mode.

In summary, we have been able to see how the NAS Grid Benchmarks scale when more processors are used, but depending on the available parallelism in the benchmark. If the maximum available parallelism in the data flow graph generated for the application is 3, we will not be able to reduce the execution time of the application introducing more than 3 machines to be used in the computation. Besides, we have seen how the file transfers are minimized between the machines forming the Grid, due to the locality policy implemented in the runtime. Finally, the performance analysis made has shown us the importance of the granularity of the task when it must be run in the Grid, as the middleware may introduce overhead in the execution. For example, having 30 seconds of overhead in a task that lasts

1 hour is reasonable, but it is not for a task that lasts 5 seconds. Thus, the middleware overhead is one of the main factors to determine if an application is suitable to be executed in the Grid.

4.3.2 Applications performance

Simple optimization example

This first case to study the scalability is an example of a simple optimization algorithm written with the GRID superscalar paradigm. The code of the master program is shown in Figure 4.19. In this example, a set of N parametric simulations performed using a given simulator (in this case, we used the performance prediction simulator Dimemas [59]) are launched, varying some parameters of the simulation. Later, the range of the parameters is modified according to the simulation results in order to move towards a goal. The application runs until a given goal is reached.

```
GS_On();
while(!goal_reached() && j < MAX_ITERS)
{
   getRanges(Lini, BWini, &Lmin, &Lmax, &BWmin, &BWmax);
   for(i = 0; i < ITERS; i++)
   {
      L = gen_rand(Lmin, Lmax);
      BW = gen_rand(BWmin, BWmax);
      filter("bh.cfg", L, BW, "bh_tmp.cfg");
      dimemas_funct("bh_tmp.cfg", "trace.trf", "dim_out.txt");
      extract("bh_tmp.cfg", "dim_out.txt", "final_result.txt");
   }
   getNewIniRange("final_result.txt",&Lini, &BWini);
   j++;
}
GS_Off();
```

Figure 4.19 Source code of the simple optimization example written with GRIDSs

Each called function performs the following operations:

- getRanges: from an initial latency value (Lini) and an initial bandwidth value (Bini), this function generates a range for L values (from Lmin to Lmax) and another range for BW values (from BWmin to BWmax).

- gen_rand: generates a random value between the two values passed. It is used to generate both L and BW parameters.

- filter: substitutes two parameters (latency and bandwidth) of the configuration file bh.cfg, generating a new file bh_tmp.cfg.

- dimemas_funct: calls the Dimemas simulator with the bh_tmp.cfg configuration file. The file trace.trf is the input tracefile and the results of the simulation are stored in the file dimemas_out.txt.

- extract: gets the result of the simulation from the dim_out.txt file and stores it in final_result.txt file.

- getNewIniRange: from the data in the final_result.txt file generates the new initial values for parameters L and BW (Lini and BWini respectively).

The interface definition file for this example is the one shown in Figure 4.20. Although the only computational intensive function is the one that calls to the Dimemas simulator, we have added the filter and extract functions too in order to have a more complex data dependency graph. Functions getRanges, gen_rand and getNewIniRange do not appear in this file because they will be run locally on the master. Although they are local functions, they can interact with the GRID superscalar runtime, as it happens in getNewIniRange function: the file final_result.txt has to be opened, thus the GRID superscalar functions GS_FOpen and GS_FClose must be used.

```
interface OPT {
  void filter(in File reference CFG, in double latency,
     in double bandwidth, out File newCFG);
  void dimemas_funct(in File cfgFile, in File traceFile,
      out File DimemasOUT);
  void extract(in File cfgFile, in File DimemasOUT,
     inout File resultFile);
};
```

Figure 4.20 Task and parameters definition for the simple optimization example

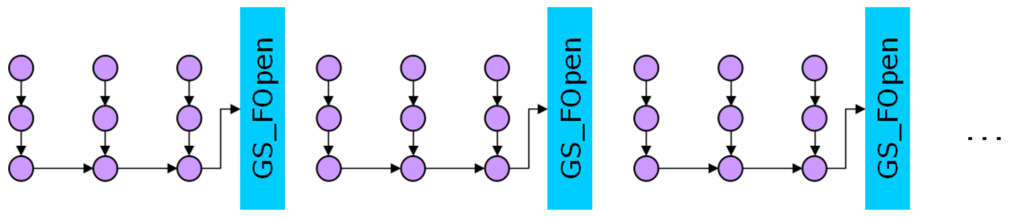


Figure 4.21 Workflow generated for the simple optimization example

Figure 4.21 shows the data dependency graph generated if we set the ITERS parameter to 3 and the MAX_ITERS parameter to a minimum of 3. We can see that the maximum parallelism achievable is 3 (determined by ITERS), and that the extract operation generates a chain of data dependencies in the inner loop. The GS_FOpen used in the getNewIniRange function causes a synchronization point: the open call must wait until the loop has finished its execution.

We have run this simple application setting the MAX_ITERS parameter of the application to 5 and the ITERS parameter to 12, leading the application to a maximum parallelism of 12, while the total number of remote tasks generated is 180. Some results of this example are shown in Table 4.6. As in the NAS Grid Benchmarks case, two different machines were used as workers: Khafre, an IBM xSeries 250 with 4 Intel Pentium III, and Kadesh8 a node of an IBM Power4 with 4 processors. The master was located in a different machine, a regular PC.

Table 4.6 Execution times for the simple optimization example

Machine	# max tasks	Elapsed time
Khafre	4	11 min 53 s
Khafre	3	14 min 21 s
Khafre	2	20 min 37 s
Khafre	1	39 min 47 s
Kadesh8	4	27 min 37 s
Kadesh8	3	28 min 27 s
Kadesh8	2	34 min 51 s
Kadesh8	1	48 min 31 s
Khafre + Kadesh8	4+4	8 min 45 s
Khafre + Kadesh8	2+2	15 min 11 s
Khafre + Kadesh8	1+1	24 min 33 s

Column *Machine* describes the worker or workers used in each case, column *# max tasks* describes the maximum number of concurrent processes allowed in each worker and column *Elapsed time* the measured execution time of each case. The number of tasks in each worker was set to a maximum value of 4 since the nodes we were using have 4 processors each. For the single machine executions, it is observed that the execution time scales with the maximum number of tasks, although it has a better behavior in worker Khafre than in worker Kadesh8.

When using both workers, we obtained execution times between the time obtained in Khafre and the time obtained in Kadesh8 with the same number of tasks. For example, when using 2 tasks in each worker, the elapsed time is between the elapsed times obtained in Khafre and Kadesh with 4 tasks. In this case, from the 180 tasks executed in the benchmark, 134 were scheduled on the faster worker (Khafre) and 46 in the slower one (Kadesh8).

We show the absolute speed up obtained in the different executions of the application in Figure 4.22 using a single machine and in Figure 4.23 using both machines. The sequential times used as a reference to calculate the speed up have been taken from running 60 times the Dimemas simulator in Khafre (2280 s) and in Kadesh8 (2700 s), without using GRIDSs (the fastest sequential time achievable). This way we can evaluate

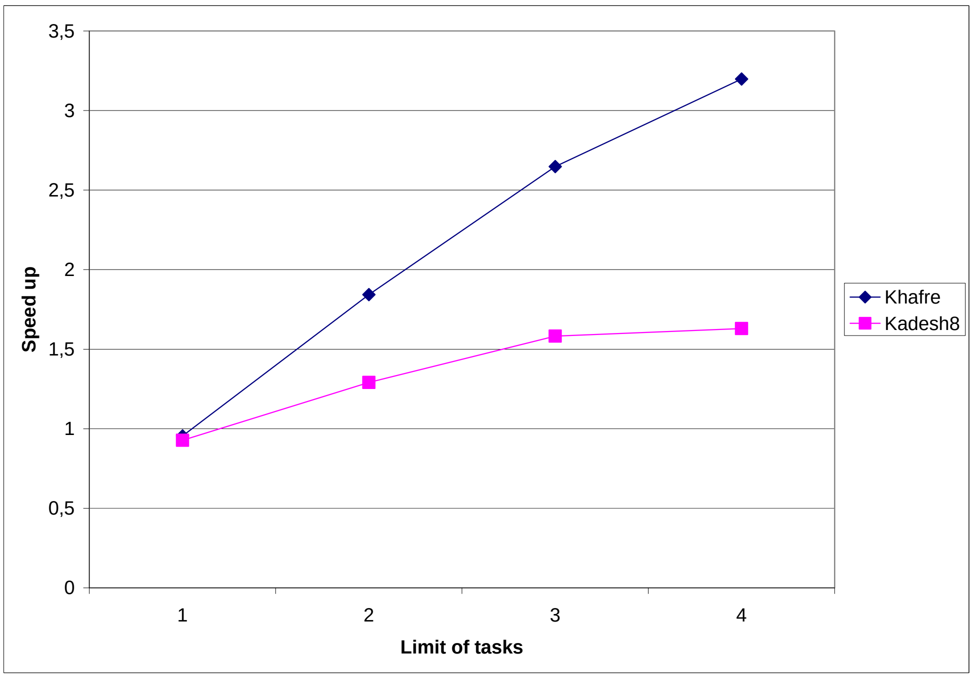


Figure 4.22 Absolute speed up obtained for the simple optimization example using a single machine

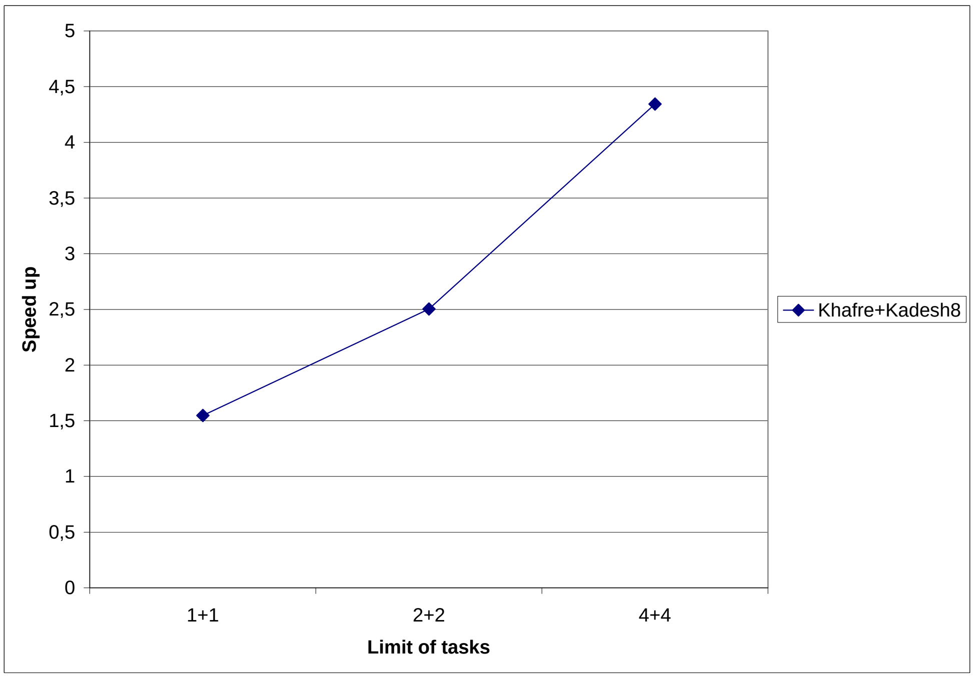


Figure 4.23 Absolute speed up obtained for the simple optimization example using two machines

if executing the application in the Grid is a good choice. In Khafre we can see that the maximum speed up reached is 3.19 when using 4 CPUs, which is good. However, in Kadesh8 the maximum speed up is 1.62 when using 4 CPUs, which is a bad result.

We can explain the poor performance obtained when Kadesh8 is involved in the computation with the conclusions we obtained in our performance analysis made in Section 4.3.1. Tasks filter and extract are immediate when executed (they do not require any computation at all) and from analyzing the execution logs we have checked that these tasks only last approximately 1 second in Khafre, but 30 seconds in Kadesh8. This is because Globus polls for the status of the task at the beginning of the run, and after 30 seconds. In Khafre the first poll always notifies that the task has finished, and in Kadesh8 the same happens in the second poll, increasing the execution time of every task.

As a conclusion of this simple experiment, we can see again the importance of selecting tasks with enough granularity to be executed in the Grid, so the Grid middleware does not kill the performance that could be achieved by using GRID superscalar to run the application.

New product and process development

In the framework of the BE14 experiment, as a working package of the BEinGRID [21] project, an application is presented that integrates the computation stages of a high-throughput environment for product and process development. The experiment allows the integration of models for optimization and simulation providing a flexible environment on top of GridAD (a tool composed by the integration of GRID superscalar and GridWay [49]). As a result of using the Grid, the performance of the experiment is dramatically increased. The high-throughput environment involves generic stages common to a variety of industrial problems, product synthesis applications, process design, materials design, and high-throughput experimentation in specialties, pharmaceuticals, and high-value chemicals. Such problems involve multiple runs, each using available physico-chemical and economic data, to target and screen options for products and processes. The combined use of computer and experiments is seen as the future environment for the development of novel products and processes.

In the specific experiment a process and catalyst development problems are modelled mathematically and solved with a combination of stochastic algorithms, deterministic algorithms, and graph-based methods. Among others, the application inputs contain (see Figure 4.24) superstructure models, kinetic data, configuration seeds and solver controls. The stochastic search takes the form of a Tabu search with parallel steps for intensification and diversification. In each step of the Tabu search, *m* different initial solutions goes

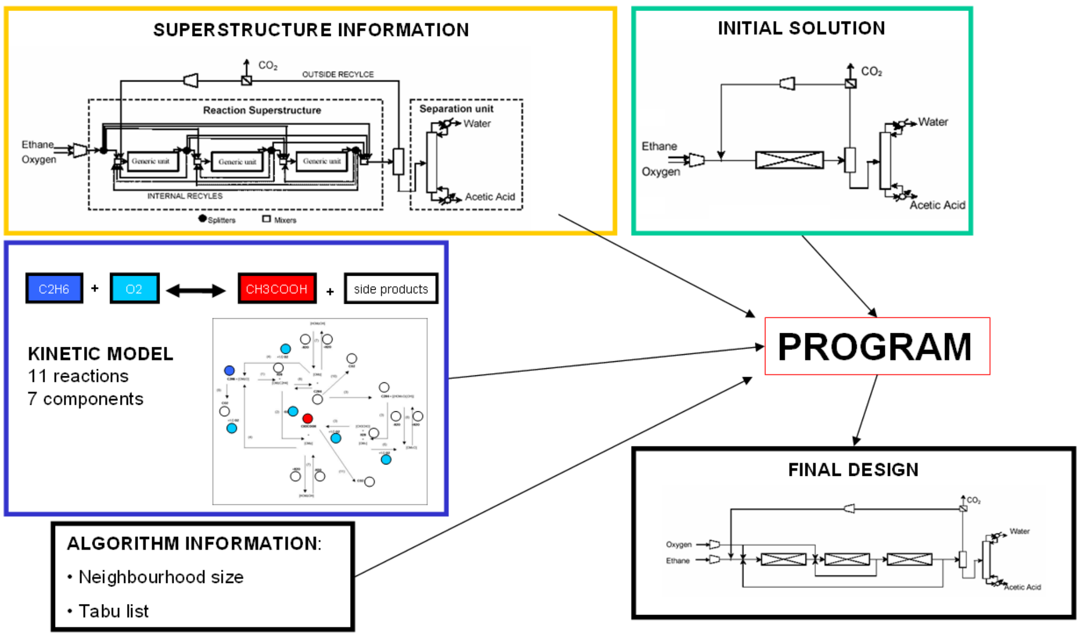


Figure 4.24 Chemical process development diagram

Table 4.7 Summary of processes

Application case	Van de Vusse	Biocatalytic	Acetic Acid
Number of initial solutions (m)	10	6	6
Number of slots (s)	50	20	50
Number of tasks (N)	2,000	480	1,200
Number of iterations per slot (i)	5	5	35
Neighborhood size (h)	7	20	50
Total number of simulations (n)	52,500	36,000	157,500

through *s* slots, and in each slot 4 tasks are executed. Therefore, the number of tasks in one slot is *m* x *s* x *4*. The system then updates the solutions with the best results, and the process is repeated *i* iterations and *h* times for a neighborhood factor. The total number of simulations is calculated as *m* x *s* x *3* x *i* x *h* since one of the 4 tasks previously mentioned is to select the best results and no simulation is performed.

Table 4.7 presents a summary of statistical information for three different processes: a Van de Vusse kinetic scheme, a catalyst design experiment for acetic acid production, and a biotechnology (biocatalytic) process with excessive requirements for computing. The number of tasks and simulations required for each experiment are summarized in the table. Real-life applications would require multiples of such simulations (typically by 3-5 orders of magnitude).

The workflow generated in the application is shown in Figure 4.25. The parallelism is determined by the number of initial solutions used in the execution. From the initial solution a first simulation is performed (which represents a GRIDSs task) and then the intensification and diversification steps are done, calling to two new simulations. This leads to a maximum degree of parallelism in the application of *2 x m*.

The Grid used for this application is composed by 3-site machines (UCM in Madrid, BSC in Barcelona and UniS in Surrey) with up to 20 workers. Preliminary execution results are shown in Table 4.8. We have used 5 machines with 8 workers. In the Van de Vusse execution, the granularity of the tasks is so small than executing it in the Grid is not a good option. The Biocatalityc run achieves a speed up of 4.47 and the Acetic Acid run 3.64. This is lower than expected, as the maximum parallelism of the graph in both cases is 12, and the minimum is 6. However, the synchronizations in every execution slot and the difference in the execution time of the tasks which must be synchronized makes the performance to be lower than expected.

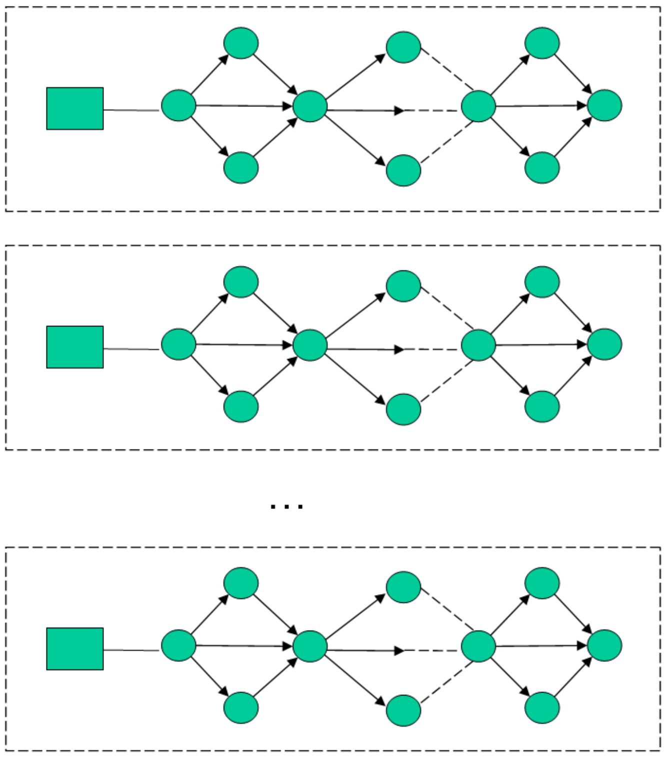


Figure 4.25 Generated workflow for the chemical process development

In this example we can see the importance of the data dependency graph to achieve a good performance. The way the application is implemented makes the parallelism depend on the number of initial solutions used as inputs: in some parts of the algorithm the parallelism available is *m*, and in some other parts it reaches *2 x m*. Thus, we cannot expect as many speed up as in a complete parallel dependency graph. Moreover, the synchronization points needed in the application also hold down the achievable performance. However, the use of the Grid in this case has enabled an easy way of externalizing the end users' computing needs, which was not possible without this technology.

Table 4.8 Execution results of the chemical processes

Application case	Van de Vusse	Biocatalityc	Acetic Acid
Sequential execution time	00:09:22	14:40:46	07:40:14
Grid execution time	00:48:46	3:17:18	02:06:33
Speed up	0.19	4.47	3.64

Potential energy hypersurface for acetone

The computational chemistry group from Universidad de Castilla La Mancha (UCLM) has worked tightly with us to port their work to GRIDSs. They determine the molecular potential energy hypersurfaces from the results of electronic structure calculations [79]. The application is tested generating potential energy hypersurfaces for the double methyl rotation and the coupling with the CCCO frame in acetone. The results predicts a barrier to methyl rotation that matches with the one obtained experimentally by different techniques. The mapping of potential energy hypersurfaces is done by first generating a large set of different molecular structures, submit them to the Grid for execution through an electronic structure package called GAMESS [38], filter and collect the results.

The GRID superscalar implementation of the application is very simple, as only a loop is needed to iterate through all input files. The task defined to be executed in the Grid is mainly composed by a call to the GAMESS simulator, which performs the needed calculations. Every call to the GAMESS simulator is independent from the rest, thus the data dependency graph for this application does not have any data dependency (it is completely parallel), which is very suitable to achieve a good speed up when it is executed in the Grid.

The calculations have been performed on a computational Grid formed by five nodes. The first, Sofocles, is the Grid master. It is a single-processor machine AMD-2.4 GHz from the QCyCAR research group at Ciudad Real, Spain. The next two are two PC clusters of 12 nodes each, also from the QCyCAR research group. The first, Tales, is a cluster formed by 12 single-processor Pentium IV, 2.4-3.0 GHz. The second (Hermes) is similar, but consists of Pentium IV, 2.6-2.8 GHz machines. The third node is formed by the Kadesh supercomputing system at the Barcelona Supercomputing Center, Spain. Kadesh is a cluster formed by 128 Power3/375 MHz + 32 Power4/1 GHz nodes. The fourth node is a cluster (Popocatepetl) of nine biprocessor 64 bits AMD machines 1.4 GHz, placed at the Laboratorio de Química Teórica in the Universidad de Puebla, Mexico. Thus, the Grid implies a transatlantic connection. The total number of CPUs used in the test is 64, distributed in 22 from UCLM, 28 from BSC and 14 from Universidad de Puebla.

Two different Grid configurations have been used in the tests. The first one uses eight workers (processors) in each cluster of the Grid. Thus, 32 processors are used, distributed as follows: eight on Tales, eight on Hermes, eight on Kadesh and eight on Popocatepetl. The second configuration uses a total of 64 processors: 11 on Tales, 11 on Hermes, 14 on Popocatepetl and 28 processors on Kadesh. The results for both cases are collected in Table 4.9. We have calculated the speed up as the absolute speed up,

Table 4.9 Execution results calculating the potential energy hypersurface for acetone

Machine	# tasks (32 CPUs)	# tasks (64 CPUs)
Tales	310	223
Hermes	290	218
Popo	286	262
Kadesh	234	417
Total time (min)	1875	1025
Speed up	26,88	49,17

considering the sequential execution of the application in the fastest machine in the pool (50400 minutes). The results are very good (despite the heterogeneity of the system) due to the easy parallelization of the application, as no data dependencies exist between the tasks.

From GRID superscalar's point of view, one of the key objectives achieved in this experiment is to have performed a long duration experiment in an heterogeneous Grid with a transatlantic link. Besides, we have analyzed how our system has been able to achieve excellent speed up results for this application.

Metagenomics

The Computational Genomics group in Barcelona Supercomputing Center has carried an analysis of protein and function diversity on earth. The identification and classification of new and known proteins is used in different areas. In the study of the Evolution it helps to understand the mechanisms of evolution of proteins and organisms. For Ecology it allows to identify new pathways and organisms for new possibilities in biodegradation. In BioMedicine, new microorganisms, pathways, proteins/molecules and new therapeutic molecules (antibiotics, protein targets) can be identified also. And finally, for Molecular Biology it provides new functional associations and hypothesis for basic research. More precisely, they have been able to run the largest protein comparison and classification done so far, by using the MareNostrum supercomputer and GRIDSs. The comparison has been done using 15 million protein sequences and the BLAST simulator [18].

The protein comparison algorithm is very suitable to be parallelized, as the comparisons are performed in an all-against-all basis. This means that no data dependencies exist between each comparison, thus the graph generated by GRIDSs for the algorithm will be a set of completely independent tasks. The algorithm is a nested loop which iterates between the two sets of proteins to be compared.

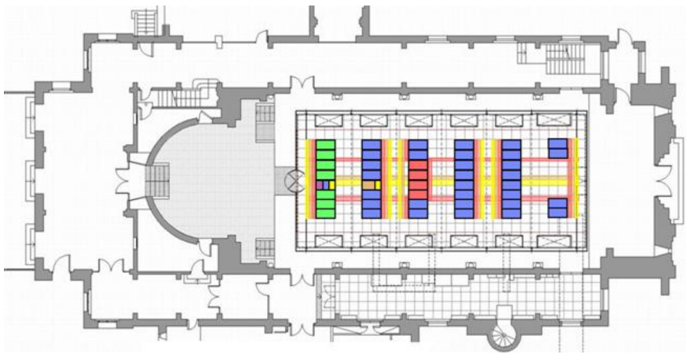


Figure 4.26 MareNostrum floor diagram

The MareNostrum supercomputer has been used to execute this test. MareNostrum is a supercomputer based on PowerPC processors, the BladeCenter architecture, a Linux system and a Myrinet interconnection. It is composed of 10240 IBM Power PC 970MP processors at 2.3 GHz (distributed in 2560 JS21 blades), with 20 TB of main memory, 280 + 90 TB of disk storage, and a peak performance of 94,21 Teraflops. Figure 4.26 shows the layout where the supercomputer is located.

As we have stated previously, GRIDSs has been ported to different Grid middlewares, but we also have a version which works inside clusers [9]. We have used this version in order to run the experiments in MareNostrum. Conceptually, a cluster can be compared to a Grid, as the nodes could be considered as independent machines. However, no data transfers are needed, as there is a shared file system which allows to read or write any information from any node in the cluster.

From the total number of CPUs in the supercomputer, 4000 have been used exclusively to run this application using GRID superscalar. Four different runs have been performed with the application, using different sequences as inputs. In each run, GRID superscalar has generated and handled more than 100000 tasks in 1000 nodes (which in the runtime are considered as a "Grid" of independent machines).

Several conclusions can be extracted from the execution of this application using GRIDSs. GRIDSs coordinates automatically the execution of all sequence comparisons, avoiding the users to implement scripts to directly work with queuing systems, because submitting such a high number of independent jobs in a queue system could cause performance problems in it. Besides, we have tested our runtime in a very demanding environment, with a "Grid" of 1000 machines which have a very low latency between them. This is a very good test to prove that our runtime performs well in such stressful situations.

SSH GRIDSs vs PACX-MPI

The fastDNAml algorithm is a commonly used code for maximum likelihood phylogenetic inference [69]. More specifically, maximum likelihood methods are very effective when the input data set includes highly divergent sequences, which increase a lot the computation required. These sets of sequences are groups of genes, gene products or taxa (groups of organisms with evolutionary relationships), from where the tree with highest overall likelihood is selected using an heuristic approach.

The MPI version of fastDNAml was executed in the SC2003 HPC Challenge context, in a project that won the Most Distributed Application award [93]. Between the Grid-enabled MPI approaches it is found PACX-MPI [72] and MPICH-G2 [56], among others. In the mentioned event they used PACX-MPI as the Grid-enabled MPI library. The idea of implementing a GRID superscalar version of fastDNAml was coined in that event.

The GRIDSs implementation of the fastDNAml [29] was created from the sequential version of the fastDNAml code and submits the evaluation of the trees to the Grid, which is the most computational intensive function in the code. In order to increase the performance of the execution, two optimizations were applied: tree clustering and execution of initial evaluations in the master. In the tree clustering, three policies were set in order to tune the granularity of tasks in every iteration step: FAIR (divide the number of trees to evaluate by the number of slots defined in the workers), UNFAIR (the maximum number of tasks generated is the same as in the fair policy, but if there are less, we pack them with the size of the pre-defined variable MAX_PACKET_SIZE) and DEFAULT (a task is a cluster of exactly MAX_PACKET_SIZE trees).

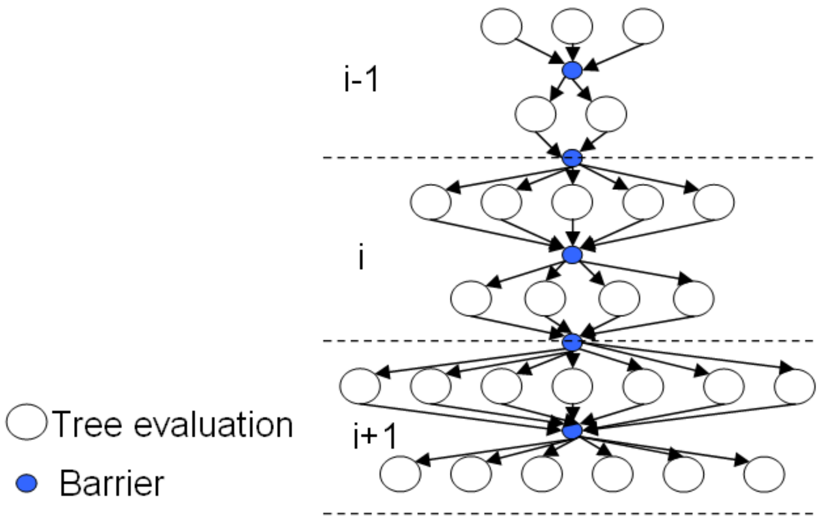


Figure 4.27 fastDNAml workflow generated by GRIDSs

The general structure of the sequential application is the following: an initial tree is created using three taxa chosen randomly. The overall likelihood value for this initial tree is evaluated. Next, a new taxon is randomly selected. There exists *2i - 5* different possible trees (with *i = 4* in the first iteration) when adding this taxon. Therefore, *2i - 5* evaluations are performed, which are independent between them. Next, a local arrangement step is performed. In each iteration, this step will generate *2i - 6* evaluations (also independent between them). This local arrangement step may be repeated while any improvement is find. Next, a new taxon is added and the evaluation continues as explained before. The process will finish when all taxa have been added. This algorithm leads to a data dependency workflow as shown in Figure 4.27.

Although initially some executions were performed using Globus as the underlying Grid middleware (see [29]), we also have available an implementation of the runtime using *ssh* and *scp*. This version allows the users to get benefit of GRID superscalar but getting rid of the Globus related problems and overheads when smaller granularity tasks are to be executed. Authentication is granted through the use of *ssh-keygen*.

Table 4.10 shows the results obtained with fastDNAml using the HPC Challenge data set in a Spanish Grid, composed of machines at two sites in Madrid (rediris.es and ucm.es) and one in Barcelona (upc.es) (Pentium III and Pentium IV based). Madrid and Barcelona are at about 500 km of distance. For each site it is indicated the number of workers used. The results show that the ssh GRID superscalar version outperforms the Globus version, for a similar number of workers and even using a larger distance Grid. Also, this version is compared with a PACX-MPI execution. It can be observed that the results obtained with GRID superscalar outperform the PACX-MPI version too.

The advantages of using GRID superscalar against MPI are mainly two. At programming level, GRID superscalar has a simpler user interface, because the parallelism is implicitly extracted from the application. At running level, MPI requires all processes to be active during all execution. This means that if the application uses 100 workers, it requires all of them available for the application from the beginning to the end (physically requires grabbing the 100 processors). With GRID superscalar each application task is

Table 4.10 Execution times for fastDNAml

Middleware	upc.es	rediris.es	ucm.es	Time
Globus	10	0	0	14235 s
PACX-MPI	1	1	7	9240 s
ssh	1	1	7	7129 s

executed as a single new program, activated by its runtime through the Grid middleware. If at a given moment the application has only 3 concurrent tasks, only 3 processors will be required. It may also use a queuing system to submit the individual tasks, using free slots in a cluster when available.

We have seen in this and previous examples that Globus is not suitable for applications which have a small granularity, because it adds a considerable overhead which may kill the application performance. In this example we have been able to present the execution results of GRIDSs using ssh instead of Globus, and the numerical results show that the ssh version is faster than the Globus version and even also faster than the PACX-MPI version of the algorithm.

Chapter 5

Applying fault tolerance at the programming model level

Some research and commercial products use the Grid as their basic infrastructure. However, because of its widespread nature, heterogeneity and dynamism, the Grid is an environment prone to failures: an execution can use hundreds or thousands of machines distributed all around the world, in different administrative domains, with different computing powers and architectures, and they may enter or leave the Grid infrastructure dynamically. It is not realistic to think that in such a widely spread environment errors will not occur. The ability to face such errors is called *fault tolerance*.

A failure is not only a software or hardware crash, but also that it does not accomplish some requirements (i.e. a performance degradation in a machine). Besides, different types of failures exist: transient (appear once and disappear), intermittent (appear and disappear randomly) and permanent (when they appear the only solution is to replace the faulty component). Different strategies can be used to deal with these different types of failures.

This chapter details our efforts to deal with failures when using GRID superscalar. The first section presents the checkpointing mechanism designed to save the work done before a failure appears. However, we also establish the objectives of being a transparent mechanism for end users, and cause minimum overhead to the execution, to follow with the main principles of our tool. It also presents an experimental execution to evaluate if our design accomplishes the objectives set.

The second section explains the retry mechanisms implemented in our runtime in order to deal with Grid level failures without having to stop the application. Another important objective is that we want to detect performance degradations in tasks to avoid delaying the whole execution, as well as the basic objectives of keeping the mechanism as transparent as possible to end users and not to cause overhead when no failures are detected. A Monte

Carlo algorithm is executed to see how we are able to keep running applications despite failures.

Finally, the last section sets as new objectives to hide the time required to resubmit a task in case of failure, and to migrate tasks to faster resources when they become available without demanding the user to implement an intra-task checkpoint (always keeping these new mechanisms invisible to users). This is accomplished with a task replication mechanism, that is tested using the fastDNAml algorithm demonstrating how replicas can help not only in dealing with failures but also in speeding up the application.

5.1 Checkpointing

5.1.1 Design

The main objective when studying the design of a checkpoint mechanism for GRID superscalar is to include the ability of saving the work performed by an application before an error occurs. This is specially important when the error detected is an unrecoverable error, which makes mandatory to stop the application because a user intervention is needed in order to fix it.

In Section 2.4 we have seen the approaches taken by other research groups in order to provide checkpointing features for applications. Their main focus are two: checkpoint a process (system level checkpoint) and checkpoint the whole application (application level checkpoint). The former can be transparent for end users, but it may become more complex to apply if the application is not formed by a single process. The latter is in most cases implemented as an API to allow users to implement the checkpoint by themselves (saving the state of the application so it can be restarted later), which means that the mechanism is not transparent. Our objective is to make an application level checkpoint completely transparent to end users.

Another important objective that we establish is to cause the minimum overhead to the application when the checkpoint mechanisms are running. The design of the GRIDSs runtime is focused to speed up applications by running them on the Grid, thus the checkpoint mechanism must not have a negative influence in the performance when no failures are detected during the execution.

When deciding a checkpointing strategy for GRID superscalar, different possibilities were taken into account:

- Leave the checkpointing actions open to the application developer, i.e., do not offer checkpointing features with GRID superscalar at all.

- Offer a traditional checkpointing strategy, which is able to restart a single process at any point.
- Tailor the checkpoint strategy to GRID superscalar, taking advantage of the specific characteristics of the applications.

The first option was discarded, since it does not fulfill the main objective of keeping the mechanism transparent to the user. Also, we did not want to develop a checkpointing strategy of the second type, since it will duplicate efforts already done by other research groups (some can be found at [22]). Implementing a checkpointing strategy of the third type will allow adding an interesting feature to GRID superscalar runtime while its complexity and overhead for the application may be kept very low.

The checkpoint mechanism included in GRIDSs is one of our first steps to achieve tolerance to failures in the system. Tasks that finish their execution correctly are saved to avoid repeating them. If a failure is detected, the application can be safely stopped. This allows users to correct the error (if their interaction is needed for that purpose) and to restart the application from the point where it stopped.

The failure detection is done by means of failure notifications from the middleware and POSIX/UNIX signal reprogramming in the tasks. The middleware notification commonly includes the reason of the failure, which is notified to the user. In case of signal reprogramming, if any of the worker tasks fails due to an internal error (zero-division, memory problem) or because it has been killed for any reason, the new routine which attends the signals would build and send a message to the master with an error code specifying the signal number received in the worker. Also, if GS_Off(-1) is used in the master, the runtime stops the application in the same manner as when a worker task failure is detected. When stopping the application, the runtime also cleans all temporary files created in the worker directories, leaving them as they were before the application's execution.

As there exist already many packages for checkpointing a single process, our focus has been to checkpoint the tasks that form the application as atomic parts of the application. Checkpoint in GRIDSs works at an *inter-task* level: the task is only saved if its execution has finished correctly. Correctly means that no application related unrecoverable error has been reported by the user for that task, and no failure in the job execution has been detected. The runtime will track the tasks that have finished correctly, and in case any failure occurs, the application may be restarted from the last task that has finished correctly.

Tasks which fail during their execution will not be saved and will be restarted from the beginning. This may be a problem when some tasks have a very long run time, or even when the application is composed of a low number of tasks. In such cases users may implement a complementary checkpoint inside the tasks by themselves or with an existing checkpoint package. Both, our inter-task checkpoint mechanism and the one implemented by the user at a intra-task level may be used at the same time in an application.

In our strategy, we assume that we can only checkpoint a task if all its predecessor tasks in a sequential order have finished. This sequential order is determined by the sequential execution of the application's main program: the master part of the application, which contains the main algorithm used to generate the tasks' workflow (see Section 4.1.1). This assumption reduces the complexity of writing a checkpoint because we do not need to write all data structures in memory to disk, and it also reduces the number of files that we must store in order to restart the application correctly. With the file renaming technique applied in the runtime to increase the parallelism of the workflow (see Section 4.1.2 for more details), a file can have a lot of different versions of itself, and in the worst case, we would need to store all of them in order to be able to restart from a specific point. This mechanism restores what we call the *sequential consistency* of the application: it stops the application as if a sequential application was stopped.

The drawback of keeping the sequential consistency is that we may not be able to checkpoint some tasks that have finished their execution correctly because some of their predecessors have not finished. For applications with a high degree of parallelism it may happen that, in case of failure, a relative large amount of tasks must be repeated on restart although they have finished correctly before. However, we reduce this effect by influencing the scheduling decisions in the runtime (giving more priority to tasks with a smaller task number, thus, prior in the tasks generation). Users may take this drawback into account when deciding the granularity of the tasks that form their application, so the portion of work lost is "acceptable" for them, or they may include a inter-task checkpoint. Once a failure is detected, the runtime waits for running tasks that have the possibility of being checkpointed. Tasks which cannot be saved are cancelled to avoid wasting time with them.

We can see an example of this behavior in Figure 5.1. Tasks 0 and 1 have been previously checkpointed, tasks 2, 3 and 5 are running, and tasks 4 and 6 are pending. Task 3 finishes its execution but it cannot be checkpointed because it has predecessors still running (task 2). After that, we receive a notification about a failure in task 2, which means that we will not be able to checkpoint tasks with a higher sequence number. The immediate decision is to cancel task 5 because we will not be able to checkpoint it even if it executes correctly.

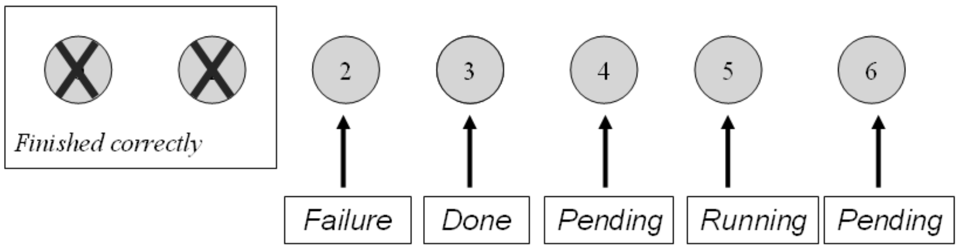


Figure 5.1 Checkpoint sequential consistency maintenance example

Several information must be saved in order to checkpoint a task. We must save enough information to be able to re-run the application skipping the tasks which were checkpointed, but without causing overhead to the system when writing a checkpoint. The first thing saved is the information of which tasks (in sequential order) have correctly finished and it is written in a checkpoint file in the master machine. The only information saved in the checkpoint file about the task is the task number, unless the task generates one or more output scalar values, which are also saved in the checkpoint file, so we do not need to re-run the task to obtain them. Figure 5.2 presents an example of the information saved in a checkpoint file, where we can see that tasks 2 and 5 have an output scalar which is stored in this file.

```
0
1
2  34.075432
3
4
5  25.093639
6
...
```

Figure 5.2 Content of a GRIDSs checkpoint file

We also need to know the file versions which are generated in the last checkpointed task and the previous ones. In case of failure, we will need to save all results generated by a task which may be considered as a final result until the moment in the execution defined by the checkpointed task (thus, the last valid version of every output file). Note that the execution moment defined by the checkpoint task usually differs from the execution moment of the GRIDSs application, as other tasks may be running to generate new versions of files. However, as those tasks have not been checkpointed, these file versions will not be considered. The saved files must be equivalent to what would have been generated in a sequential execution of the application until the last checkpointed task.

```
for(i = 0; i < 10; i++)
{
  T1(..., ..., "f1");
  T2(..., ..., "f2");
}
```

Figure 5.3 Code which causes file renaming

In Figure 5.3, we present a simple example to understand which is the file version saved in case of failure. This algorithm generates 20 tasks which can run in parallel (T1.0, T2.0, T1.1, ...), and for that purpose the destination files "f1" and "f2" are renamed 10 times each (i.e. f1.0, f1.1, ...). If task T2.5 causes a failure, we could think that the last valid versions of the file may be f1.5 and f2.4. This would be true if all tasks prior to T2.5 have finished their execution and those files have been generated. However, if the last task that can be checkpointed is T2.3, the last versions would be f1.3 and f2.3.

As in the moment we stop the application only a single version of a file exists, the renaming can be safely undone. This is very useful for end users, as they may check their results before the application is stopped. In case we had decided to consider as correctly finished all tasks (not only the ones which accomplish the sequential consistency) it would have been necessary to keep all different versions of renamed files, together with the information of which file correspond to each task, which is the original name of the task, which are their other versions, and so on (thus, almost all the data structures of the runtime). Moreover, it would not be possible to undo the renaming applied in the files.

All the files locally opened with the read option by the master (GS_Open and GS_FOpen) must be kept to allow a correct restart: the instructions executed after the open must read the same values read previously. This is not only important to keep results consistent, but also because it may happen that information read from the file is used to take decisions about the task flow order or used as parameters by subsequent tasks. We may find a case where the files which are opened by the master are very large. Since a copy for each version of the file is stored, the disk space required will grow. We consider that this would not be the case in most of the applications, but if it becomes critical, the code involved in the opening and reading of these files could be encapsulated in a remote task, eliminating the problem. The increased disk usage in the master corresponds to the addition of the size of all the files referenced in a GS_Open call in the code, using a read or append mode. However, as this call means a synchronization point in the master code, it is very unlikely to be massively used in the code. Our experience working with GRIDSs confirms this trend.

Despite the drawback of an increased disk usage in the master, one of the key features of this checkpoint mechanism is that users do not need to make a programming effort to enable their applications with this mechanism, as it is completely transparent to the application's code. In order to restart an execution which has failed, users only need to re-run their main program. If a checkpoint file is found, the runtime will start automatically from the stored point. If there is no checkpoint file, the execution will start from the beginning.

Another key feature is that no overhead is introduced at run time, because we neither save the runtime data structures to disk periodically, nor save all different versions of a file. Only the task number is saved for the task that has finished correctly. Therefore, the checkpoint has no negative effect on the performance of the application. The lack of overhead also contributes to the transparency of the mechanism, since it has no negative effects on the performance. Thus, users do not even have to enable or disable the mechanism to achieve the maximum performance for their applications (it is always enabled). Moreover, the application can be safely restarted as many times as required if future failures are detected.

Finally, the checkpoint is a technique which is useful for any type of failure detected in the application: application level errors and Grid level errors.

5.1.2 Experimental results

In order to demonstrate that our checkpoint mechanism does not introduce any significant overhead to the runtime, we have run an experiment with our simple optimization algorithm. This algorithm has been already presented in Section 4.3.2, and it is composed of a set of parametric simulations using the Dimemas simulator. For this experiment We have set the MAX_ITERS parameter of the application to 10 and the ITERS parameter to 3, leading to a data dependence graph as the one shown in Figure 5.4. The maximum parallelism of the graph is 3 and the total number of tasks to be executed in the Grid is 90.

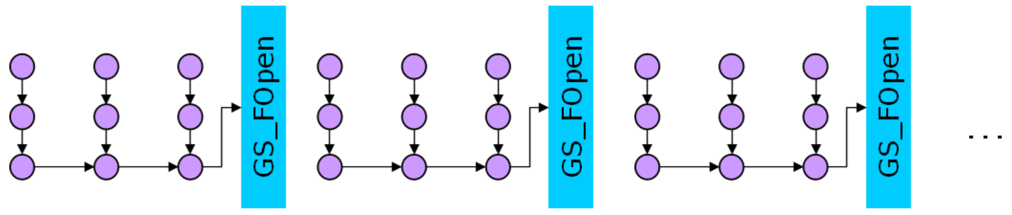


Figure 5.4 Workflow generated for the simple optimization example

We have compared the execution times of the application with the checkpoint mechanism enabled and disabled. Besides, we compared these execution times with the total time of the same application emulating one failure, calculated as the time since the application begun until the failure, plus the time from restart until the application ends.

As in the previous test using this algorithm, two different machines were used as workers: Khafre, an IBM xSeries 250 with 4 Intel Pentium III, and Kadesh8 a node of an IBM Power4 with 4 processors. The master was located in a different machine, a regular PC. We have used 2 CPUs from every machine to run workers because the maximum degree of parallelism of the application is 3, and no more than 3 workers will be running at the same time. The Grid middleware used is the one used in our reference implementation of the runtime: the Globus Toolkit.

Figure 5.5 compares the results of the second type of these experiments. The dots joined by the solid line correspond to the whole execution time of the application when no checkpointing mechanism is used (each dot correspond to a complete execution of the application). The execution time is measured as the total execution time of the master program. Since the master waits for all tasks to finish, it includes the whole execution of the application. The dots joined by the dashed line correspond to the whole execution of the application when using the checkpointing mechanism. Both sets of executions have been run under the same conditions: same load in the hosts, same number of machines to be used as workers and same tasks in each machine. However, between each set, the experiments were run with non exclusive access to the machines and with different system load. The different measured execution times (which are independent between them) have been ordered by time (from higher to lower values) in order to be able to compare them with the corresponding results done with the same system load. From the figure it is clear that the checkpointing mechanism does not increase the total execution time of the application (when no failures appear). The average execution time without the checkpoint is 486.67 seconds and with the checkpoint it is 487.67 seconds. So, for this application and environment the execution time overhead is less than the 0.2 %, which is very low.

Figure 5.6 compares the results of the execution without the checkpointing mechanism against the execution with the checkpointing mechanism when one failure occurs (in dashed line). To generate this second set of measures, a failure was generated (and therefore, the application is stopped). Later, the execution was restarted and completed correctly. The time shown is the addition of the execution time from beginning to failure (including the time invested to save the necessary information and files) plus the time from restart to the end of the application. The average run time in the first case is 486.67 seconds and in the second 512.27 seconds, which represents an increment of

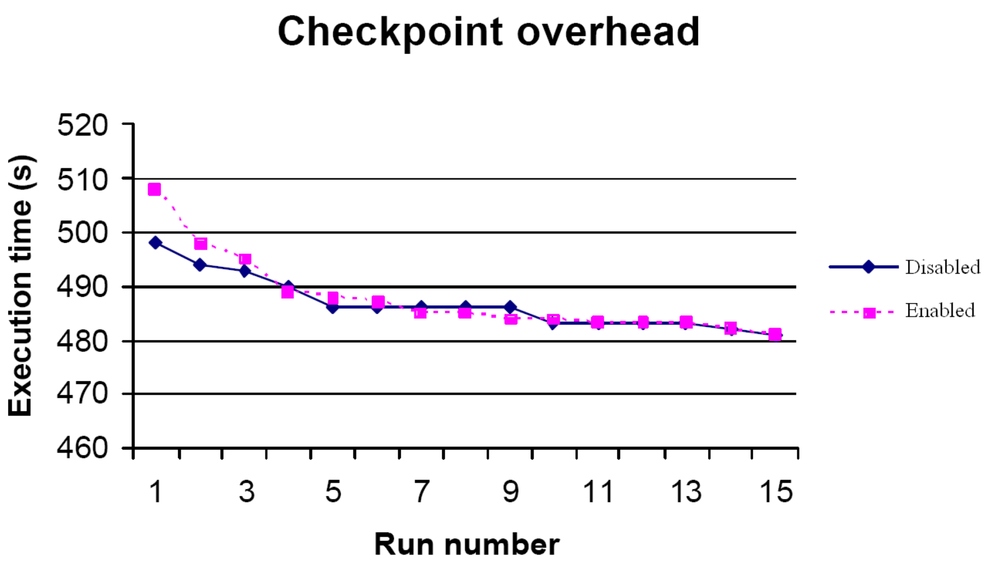


Figure 5.5 Checkpointing overhead in the simple optimization application

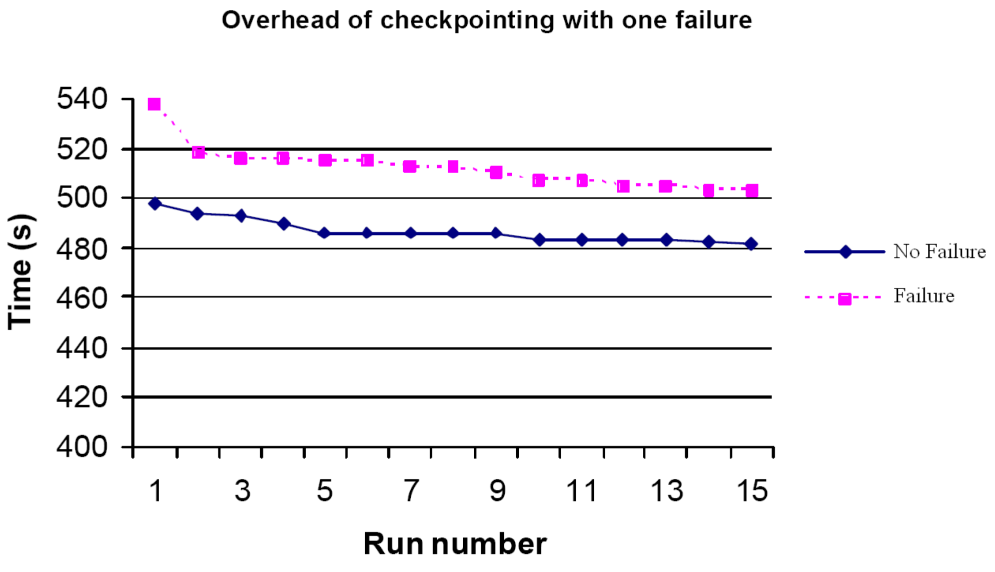


Figure 5.6 Checkpointing overhead in the simple optimization application emulating one failure

5.3%. From these extra 26 seconds, 8.4 seconds are used from the Grid middleware to detect the failure (average failure detection time) and 1.2 seconds are needed to restart the application (average restart time). The rest are run time of the task before the failure and the time needed to recover the files when stopping the application due to the failure. The main overhead in this case depends on the time the task has been running before the failure is detected, which is wasted time because no intra-task checkpoint is implemented.

In this algorithm, there exists a GS_Open with read mode inside the function call to generate_new_range. Every time a call to GS_Open has been performed, a copy of the final_result.txt file version has been made. As the parameter MAX_ITERS has been set to 10, the runtime has stored 10 copies of the final_result.txt file in the master during the execution. However, this file is small (KBytes) and those copies have not been an important overhead in terms of disk usage. In terms of execution time, the overhead is the time taken to copy 10 times the file, which is negligible.

The main conclusion we extract from this experiment is that we have demonstrated that our checkpoint mechanism does not add significant overhead to the application run time when no failures are detected. Moreover, we have checked that the application is correctly stopped and can be later restarted to finish returning the correct results.

5.2 Retry mechanisms to skip Grid level failures

5.2.1 Design

The checkpoint mechanism presented in previous section is essential when an unrecoverable error appears during the execution of an application. This type of error needs the intervention of the user in order to fix the problem. In this case, the checkpoint mechanism allows the user to correct the error and restart the application from where it was stopped. However, there exists also transient and intermittent errors in the Grid infrastructure which may be overcome without the need of a user interaction. Our objective is to treat those failures automatically at runtime, to allow applications to keep running despite any possible failures that may arise in the Grid. We want to overcome any temporal failures and to detect and skip any permanent failures.

In the Related Work chapter, more precisely in Section 2.4, we have seen that several systems implement a retry mechanism to try again the task which has failed. However, in some systems the mechanisms are not transparent to end users, and they are static (decided before running the application). Others are transparent, but they cannot tackle complex workflow applications or they are not able to detect performance degradations of the application because they do not have any application-level knowledge. In our case we want to include all the best capabilities of these different systems: handle workflow applications, transparent to users, dynamic decisions to retry tasks, and be able to detect performance degradations.

In addition, to follow with the GRID superscalar main ideas, we want the implemented mechanisms to be as transparent as possible to the end users, making them automatic or used with a very simple interface. Also, as the main goal of GRIDSs is to speed up applications executing them in the Grid, we pretend to cause the minimum overhead to the system with the mechanisms when no failures are detected, to avoid losing the performance obtained by parallelizing the code.

As application related errors could already be treated in GRIDSs (their detection and treatment can be programmed in the master program), if we are now able to overcome failures at the Grid level, we will be providing a complete solution able to deal with all possible types of failures when executing an application in the Grid.

To overcome Grid level failures, we use a *task retry* mechanism. Considering the different types of failures, transient and intermittent failures can be handled by retrying the operation which has failed, but permanent failures can only be treated by removing the machine which is failing from the pool of machines available for the computation, because we want a treatment of permanent failures which is transparent for the user. Either way, whenever the permanent failure is solved, a user may add the machine again to the computation dynamically at run time, but only if they are interested in doing so. The following sections will describe the specific implementations in order to deal with these failures in GRIDSs.

Automatic discard of machines

When a machine is unable to execute jobs due to a failure, GRIDSs must be aware of this and adapt its behaviour to the new circumstances. This situation can occur frequently, if we look at the Grid as a large collection of heterogeneous machines. The only way to skip a permanent failure in a machine without having to stop the application is to remove it from the list of available machines to run jobs, thus avoiding to use it.

Previously, in our system a user was able to manually drop a machine from the computation with the dynamic host reconfiguration script, which again reads the GRIDSs configuration file that specifies the hosts that can run jobs for a particular application (see Section 4.2.3). This solved the situation where a machine was giving a poor performance and thereby slowed the overall execution. The problem was that a remote failure in the execution made the master to stop, notifying the user of the same. Now, at run time, whenever a job execution fails, the Grid services are checked (i.e. the Globus Toolkit services [33]) and if they are running correctly the operation is retried a maximum number of times. If the check fails or the retries reach the retry limit, the machine will not be used to execute other tasks. In brief, we do not require any user intervention to deal with permanent failures in machines.

In order to discard a machine, the runtime waits for other jobs running there. This is because it may be that only the Grid job submission service fails but not the remote resource. Therefore, new submissions to the resource do not work, but previous running jobs keep executing and sending notifications to the master. Even the service for transferring files could continue running.

Because of the policy to exploit file locality implemented in GRIDSs, it may be that some result files are only available in the machine that must be dropped. As a result, before discarding the machine the runtime checks whether some files are only available in the working disk of that machine, and tries to transfer them to the master. If the file transfers also fail, the only possible solution is to restore the sequential consistency, stopping all the remote tasks, and resuming from the last checkpointed task. This will also be done automatically at run time by the library without any additional effort from the user while allowing the execution to continue.

Soft and hard timeouts for remote tasks

Many different failures may arise when an application is executed in the Grid. Some of them can be notified by the Grid middleware, however we cannot trust that it notifies all possible failure situations. Moreover, we want to consider the performance degradation of task as a failure, and to know if a task is performing well, information from the application is needed. With this objective, we use the information from the application available in GRIDSs in order to determine the best moment to detect and deal with failures for every single task. As it has been explained in Section 3.2.5, GRIDSs contains information about the estimated execution time for every task that must be run. We use this information to define two upper bounds: the timeout factor and the resubmit factor.

The timeout factor determines when the master is going to start asking about the status of a remote job. This is calculated by multiplying the estimated execution time for a task, and the specified timeout factor (its value defaults to 1.25). For instance, if the estimated time for a task is 100 seconds, a timeout factor of 1.25 means that for this task, when the execution time reaches 125 seconds, the runtime is going to check that the task is still running. If the Grid middleware responds positively, the checks will continue every 30 seconds (to avoid excessively overloading the machines, and in correspondence to Globus poll job manager interval) until we reach the time specified by the resubmit factor. If the task does not respond, or the status is not running, the task will be resubmitted in the same machine. This is done because a transient failure may have affected the task, but the machine may still be able to run new jobs. When a machine reaches a certain number of resubmits, it will be discarded and the task will be resubmitted into a new resource.

With the timeout factor the checks for tasks start only when the execution time is longer than estimated, so machines are not overloaded with status queries while the task's run time is the expected one. If the task executes normally (without any failures or any performance degradation) no overhead will be introduced to the application's total execution time.

The resubmit factor is like a hard limit for the task execution time. It specifies the time when the runtime is going to kill the task, because it considers that the execution has become abnormal. This will avoid the runtime waiting for a specific task for a long time due to performance degradation in a specific machine, thereby delaying the whole execution process, and also avoids the master hanging up when a task does so. The resubmit factor is calculated in a similar way to the timeout factor: multiplying the estimated time of the task and the resubmit factor. Its value defaults to 2, so in an estimated time of 100 seconds, the hard limit will be reached at 200 seconds. Whenever this limit is reached for a task, the task is killed and resubmitted in a new machine (not in the same machine as in the soft limit, because we consider that the machine is temporary overloaded). The resubmitted task goes to a new resource and new tasks help to check if the machine is able to run more jobs or not. However, as with the soft limit, if several tasks have reached their hard limit in the overloaded machine, we may also discard it from the computation.

The decision to use timeouts for the retry mechanism is justified because it is generic to be used with all types of applications, and does not require the user to notify to the runtime anything about the task progress. As the timeouts are based in the estimated execution time, the successful application of this retry policy depends on having an accurate estimation of the task execution time. A bad estimation would lead to a wrong application of this retry policy. Initially we leave the responsibility of estimating this time to the user, as it has been shown in Section 3.2.5, but our interface allows to easily call to an external entity which could perform this estimation instead of the user. Moreover, the timeouts can be easily disabled by users with an environment variable, in case they cannot provide accurate estimations.

Asynchronous transfers of results for each task

Our checkpoint mechanism was initially designed considering that file systems were always available in case of failure. This is because GRIDSs exploits the locality of files, and some results may only be available in a worker during the execution. However, if a task is checkpointed, but its results are in a machine that we cannot access, we reach an inconsistent state. This is also important when a machine is dropped, because the files only available in that machine must be recovered in order to reach a consistent state in the checkpoint. Of course, if a file corresponding to a task that has been checkpointed is no longer available, the checkpoint could be undone to reflect that we are not able to save its results, but we foresee that this will not be a good solution, as the checkpoint undo process could undo all the checkpoints done up to that moment, consequently not having checkpoint at all.

In order to improve the checkpoint mechanism to avoid these inconsistent states, we transfer the results of each task (once the task finishes its execution) to the master while leaving a copy in the worker as well, to keep exploiting the locality of data. These transfers are done in a background process during the execution, thereby hiding the time taken to transfer the files to the master and not delaying the whole execution. With this new mechanism, a task will be checkpointed when results are available in the master. Once a task is checkpointed, now we can ensure in all cases that the task checkpoint is not going to be undone, so the task will not be computed again.

There is still another benefit to add asynchronous transfers of results: the fact of transferring these results to the master during the execution means that, at the end of the whole execution, the post process needed to retrieve final results to the master is very short or non-existent. In brief, we are hiding the post process time with the execution time, speeding up a bit more the whole execution. Besides, we have also exploited this mechanism to undo the file renamings in these results. This allows a user to easily follow the execution results of the application from the master machine, as if the application was executed sequentially, because the files are available in the master with their original names. This is done by means of creating a soft link to the corresponding renamed file, thus no extra disk is used.

We have also identified two drawbacks: a larger usage of the master's disk and a higher usage of the network resources. This happens because previously the different versions of a file were stored in the remote workers, and only the final version was transferred to the master. If we analyze the disk usage, the master is now a mirror of all the data contained in the workers, but with only a single instance for each file. We could have reduced the disk usage in the master by delaying the transfer of files until the checkpoint must be done.

However, if a result generated some time ago in a machine becomes unavailable, this could mean that we were not able to checkpoint a task which was already executed, thus the only possibility would be to run it again, with the corresponding waste of performance. Our criteria is to give more importance to performance rather than to the disk usage, thus the benefits outweigh the drawbacks identified.

Retry of operations inside the runtime

The Grid middleware operations performed inside the runtime are also retried (a maximum number of times) to avoid temporary failures. However, different retry strategies are implemented internally in GRIDSs runtime, depending on the objective of what we want to retry. We can see an example of this in the jobs submitted by the GRIDSs post process (when the application finishes the execution). The objective of these jobs is to cleanup old versions of result files, input files, and retrieve the latest version of these mentioned result files from remote machines. In GRIDSs several machines can share a working directory, so when a job submission fails for a specific machine, a different machine with the same disk can be considered for submitting a post process job. For example, let us assume machine A and B work with the same working disk named Disk1. If we are not able to submit a job to A in order to cleanup and retrieve files, we can try the same thing with B, obtaining the same final result: Disk1 has been cleaned up. Therefore, the retries implemented in this case not only consider machines, but also disks. The same process happens when asynchronously retrieving result files of a task (as previously described).

Avoid situations that make the master stop

GRIDSs is based on a master-worker programming paradigm. Until now we have seen techniques to deal with failures in remote machines (workers in GRIDSs) but not for the master (the machine where the application is started). The way to detect failures in the master is to control errors in system calls and retry them to overcome transient failures that may appear. We have also completely revised the runtime code, and we have removed all the avoidable situations that previously made the master to stop. Now, the master dynamically resizes all predefined parameters such as GS_MAXPATH (maximum size of a path), GS_GENLENGTH (maximum size of a scalar parameter), and so on.

To conclude this section, we want to remark that our programming model addresses application and Grid errors for workflow applications. GRIDSs allows users to treat application specific errors (i.e. numerical) when the application is programmed, thus, when the workflow related to the application is generated it will include that failure treatment. The impact of unrecoverable errors (such as memory leaks) is reduced with the checkpoint technique (to restart the application from the point it was stopped). Finally, the task retry mechanism deals with errors in the Grid level, therefore all levels of failures are now handled.

5.2.2 Experimental results

We have designed a long test case as a proof of concept about the new fault tolerance features available. It is important to mention that the goal of this test is to demonstrate how the designed mechanisms work in a real case. Thus, the objective is to overcome all Grid level failures that may happen during the execution of the application. As our mechanisms are based on retrying operations, it is obvious that the more failures are detected, the less performance the application will obtain compared to an error-free execution.

The Grid for this experiment is formed with a total number of 60 CPUs, with machines from a single PC to a cluster of 29 CPUs. They are distributed in three different administrative domains around Spain (Ciudad Real, Madrid and Barcelona). These systems have Globus installed, as well as the latest GRIDSs distribution.

We have implemented a Monte Carlo algorithm composed of 7200 simulations, in order to run it over approximately 5 days. All tasks will be able to run in parallel, as no data dependencies exist between them. In this test, having a more complex workflow with data dependencies between tasks would not change our conclusions, because the retry mechanisms are applied to the tasks which are ready to run in a resource. Thus, it does not matter if the task has predecessors or successors in the workflow.

In order to emulate failures in this Grid, we have defined four different types. Each of these types includes more than one real failure situation, as we describe below:

- Kill a worker: the worker part of the application is a binary run in a remote resource. This situation may be when the associated process is killed by a machine administrator, a local resource manager (i.e. a queuing system in a cluster) or because the machine is going to be shut down.

- Grid service failure: Globus has a process that spawns and controls the running worker. We will kill this process to emulate a failure. The situations where this could happen are the same as in the previous error type.

Table 5.1 Summary of failures

Day	Machine	Type	Detection
Day 1	bscgrid02	Kill Workers	Middleware
Day 2	bscgrid03	Kill Service	Timeout
Day 3	bscgrid04	Performance	Resubmit
Day 4	bscgrid01	Stop Service	Middleware

- Performance degradation: a shared machine could experience a performance degradation because processes from other users require the CPU or almost all the memory available. This could happen in machines without any local resource scheduler, where users can submit their processes anytime. We have emulated this by changing the simulation that must be performed in the worker by a longer one.

- Grid service unavailable: we disable the job submission service to emulate that no new jobs can be submitted to a resource. A real case with the same consequences could be not only a service failure itself, but also a power or network outage in the remote resource.

The reason for not including any failures in the file transfer service is that, in Globus, the job execution service controls the file transfers of a job, thus any failure in a file transfer would be detected and notified by this service. The emulation of these failures has been implemented with scripts submitted in different days during the computation. Every error type has been emulated in a single machine, as described in Table 5.1. From the 60 CPUs used in the experiment we have caused failures only in the machines where we have administrator rights (from BSC), thus in 8 CPUs. We killed the two workers running in bscgrid02 every 4 hours the first day, the same we did with the Globus processes that control the workers in bscgrid03 the second day. The third day of failures, we changed the simulator to a longer one for 2 hours every 4 hours in bscgrid04. Finally, we disabled or enabled the job submission service every 2 hours the fourth day, letting this service half of the time available.

As we have seen in previous section, our retry mechanisms are based on the estimated execution time of the task. In this example, the cost function to calculate this estimation is implemented with information from a previous run of the simulator, as shown in Figure 5.7. With the execution time and the computing power available in those previous runs, we can estimate the time that the simulation will take in a new candidate machine. It is also important to mention that, as our estimation is accurate, we have set the timeout factor to 1.05 and the resubmit factor to 1.10 (instead of the default values 1.5 and 2

```
double Dimem_cost(file cfgFile, file traceFile)
{
  double factor, operations, cost;

  factor = 1.0486-e06; //How size of trf
     affects the number of ops generated
  operations = GS_Filesize(traceFile) * factor;
  cost = operations / GS_GFlops();
  return(cost);
}
```

Figure 5.7 Cost calculation in the Monte Carlo algorithm

respectively). This means that checks for the job will start when the task execution time is 5% higher than expected, and a job will be killed when the time is 10% higher than expected. This is done to reduce the time lost in a resubmit, as the time the simulator takes to be run in the machines where we produced failures is approximately 1 hour.

The total execution time has been 5 days 15 hours 47 minutes and 39 seconds. One of the interesting things to see in the test is the differences in error detection. When the worker process is killed, Globus detects it and notifies the GRIDSs runtime, thus the detection of the failure is immediate and the task can be resubmitted at the same moment. However, when the process that controls the worker is killed, no failure is detected by Globus and it is detected when the task reaches the soft timeout. This was unexpected for us, as we thought the Grid middleware would detect and notify this error. Anyway, our runtime has been able to detect and solve this type of failure. This case demonstrates that fault tolerance at the programming model level can protect the application from extra errors that the Grid middleware is not able to detect, or because the middleware does not implement any fault tolerance.

When the performance degradation is emulated, the hard timeout is reached and a resubmit of the task is performed. Globus does not notify any error in this case, as it does not have information about the expected execution time of the task. The last error case, when the job submission service is stopped, is notified directly from the Globus middleware, so GRIDSs can resubmit the task immediately to a new resource.

Looking at the execution logs we also see 16 unexpected retries in bscgrid01 caused by tasks reaching their hard timeout. As bscgrid01 is a shared machine without any local resource manager, other users running their processes may have caused temporary overload in the machine that has been detected by GRIDSs. However, the most important thing is that, despite all the failures registered during the whole execution (expected and unexpected), the application finished successfully.

5.3 Dynamic task replication

5.3.1 Design

The task retry mechanism presented in previous section achieves to keep an application running despite failures by means of retrying the task which has failed. However the penalty of executing the task again from the beginning must be paid when no intra-task checkpoint is implemented: the execution time of the previous task which has failed has been lost. Therefore, task retries mean that the more failures we have, the less performance we will obtain executing the application. We assume the objective of achieving the same degree of tolerance to failures but hiding this overhead caused by retrying tasks due to failures.

Besides, GRID superscalar does not implement an opportunistic task migration mechanism to migrate tasks between machines during their execution. This mechanism allows to use faster resources in the Grid when they become available by migrating the processes running in slow machines to faster ones. The reasoning why this mechanism is not included is related with the need of a intra-task checkpointing mechanism to migrate a task, which in most systems makes mandatory to the end users to implement the checkpoint of the tasks by themselves. This lack of transparency to the end users differs completely to the GRIDSs philosophy of keeping the Grid as invisible as possible to the end user. Thus, we want to investigate ways to use the fastest resources available in the Grid in a given moment, but without forcing the users to implement a checkpoint for their tasks.

The suitable mechanism to achieve both objectives is the *task replication*: to submit a task more than a single time in the system. In Section 2.4 we have seen that static replication of tasks to achieve tolerance to failures has been already used in other systems, but the replication details are decided before execution (static). Other systems use task replication to obtain redundant results and avoid malicious replies or wrong calculations in the machines which form the Grid, which is not among our objectives. In our case we want to provide a dynamic task replication, able to use application-level knowledge to decide what task must be replicated depending on the running application and the Grid used. Besides, to our knowledge, task replication has never been used to gain performance in the Grid, which is our second objective.

Executing a task in more than one machine has some advantages and drawbacks that must be discussed. One clear advantage is that, in case of a failure in the original submitted task, there are other machines already performing the same job. Thus, even with failures, the execution time of the application could be unaltered if we can get the same result from a replicated task as if it had been executed by the original one. This avoids overheads introduced when a resubmit of the task is used to try to generate correct results again.

Another advantage is related to the heterogeneity of the resources in a Grid. It may be that, after submitting a task to a machine, a more powerful resource becomes available for executing jobs. Without an opportunistic task migration mechanism, the task would be executed in the slower resource, even when a more powerful one is available. However, with task replication, we can submit a new instance of the same task in the new resource available, and we will obtain the results faster, speeding up the computation. This is very important, as we obtain results similar to the ones obtained using opportunistic migration using a completely different philosophy. Moreover, with the advantage that a user does not need to implement an application specific checkpoint for his task in order to restart it in a new resource, which may be difficult depending on the application. We will get as the final result for a task the fastest one generated from all replicas, the rest will be discarded, thus there is no need for other synchronization mechanisms for the replicas.

The identified drawbacks are related to a higher usage of the resources, because we execute a task more than once. In GRIDSs the machines are considered to execute a maximum number of tasks concurrently (what we call *execution slots*). If a task replica uses an execution slot in a machine, this slot cannot be used by other ready tasks that are to be run. In this case we could be slowing down the overall execution time achievable without using this physical redundancy described. We have specifically considered this drawback when designing a task replication mechanism for GRIDSs. We believe that increasing the usage of the resources is not a bad thing, as long as the replicas do not delay the start time of other tasks. The execution time with task replication when no failures arise must be the same or less than without using this technique.

We have designed and implemented a task replication mechanism in GRIDSs trying to achieve the advantages described and to minimize the drawbacks. The first thing to consider is how many replicas we are going to submit to the Grid in total. As one of our main goals is not to increase the execution time when using task replication in an error free environment, we have considered that we will only submit replicas if we have available slots for running tasks in our machines. So, the amount of replicas we can submit depends not only on the number of resources of the Grid, but also on the degree of parallelism of the dependency graph: if the degree is low, few tasks will be able to run in parallel, thus there will be more free slots in the Grid to submit the replicas. If we have a workflow without dependencies, there will not be any replication. The reason why we do not ignore the slot limit in the machines is because submitting more jobs than those expected in a machine will result in a stressed usage of the computing resources, thereby possibly causing a performance degradation in the tasks running in that machine. Thus, our main focus is to exploit the resources that have been assigned to our application to the maximum with the replicas, but without overloading them. This would also be suitable if our application is submitted entirely to a cluster, because all the resources are reserved only to the application. This way, we are giving a useful goal to idle functions during the execution.

The policy of using only free resources to submit replicas is also convenient to ensure fault tolerance in the sequential parts of the workflow that describes the application, which are critical to continue with the execution. If due to the workflow, we are only able to run a single task in a precise moment, and this task must be resubmitted because of a failure, we are delaying the whole execution even when we have free resources which could have not failed, as the execution will not be able to continue until this task finishes. In contrast, when many tasks are available to be submitted in parallel to the Grid, if one fails, it can be resubmitted while the rest of the algorithm keeps executing. The impact on the total execution time of the application is lower in this second case.

In order to emulate task migration with replication we may also think in sequential parts of the application or with low parallelism. If we are only able to submit one task due to the workflow, we select the fastest machine available in the pool. However, if a new machine becomes available[1] which is faster than the previous selected, we replicate the running task to try to achieve results faster. We do not consider the remaining time of the running task to replicate only if we will get results faster in the new machine, because no other tasks need the resource and the replica may still be beneficial in case the original task fails.

Our system only replicates tasks which are already running for two main reasons. The first one is that it is the right moment to decide which task must be replicated to achieve our objectives, and the second one is because the data dependencies have been already solved, so no conflicts with the workflow can appear.

If we do not have enough resources to replicate all running tasks, we need to establish a priority about which tasks are more important to be replicated. This case may be common in high parallel regions of the code, where almost all resources are used at the same time, so we must decide which of the running tasks should use the free slots with replicas. We will consider the data dependence workflow in order to decide which one of the running tasks should be replicated, thus taking advantage of the application knowledge available at this level. As the true data dependencies described in the workflow are unavoidable, we are interested in replicating the tasks that have more successors: the more dependencies

[1]Because it is new in the pool or because it has finished executing a task

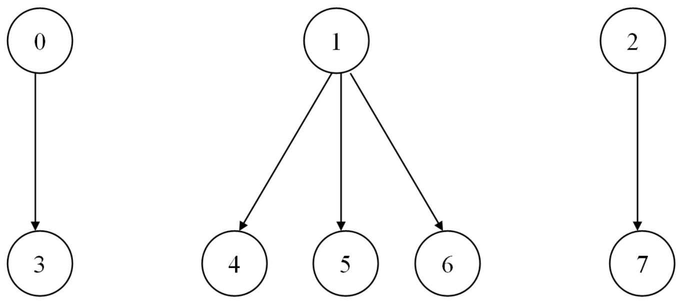


Figure 5.8 Workflow example for replication

we solve, the more tasks we will have ready to run in the next scheduling step. This is very useful when the workflow which describes the data dependencies has a low degree of parallelism: i.e. Figure 5.8 describes an example of a data dependency graph. If we suppose that tasks 0, 1 and 2 are running, we can see that, in terms of achieving a better parallelism, it is important that task 1 finish correctly as soon as possible, considering their successors, because it will add more tasks to the ready queue once it finishes. In this example, we would like to replicate task 1 to prevent a possible failure that does not allow us to free its data dependencies. According to [24], we could classify our replication mechanism as semi-active, because the decisions about the replicas are taken in the master part of the application by creating, submitting or destroying them.

The task replication mechanism also needs to determine how many replicas for a single task will be run. We have initially defined a single replica per task. The suitable number of replicas for each task is an interesting study, but it is not among our initial objectives. This will be evaluated in the future. We could consider, for instance, submitting more replicas of a task depending on how many successors the task has in the data dependency graph, and also on the free slots.

It is true that performing a deeper analysis of the dependency graph, analyzing its critical paths and estimating the execution cost of the whole workflow in every scheduling step would probably give us a more precise plan in order to decide which task should be replicated. However, as dependency graphs can be as complex as the application's source code describes, and the dynamic nature of the Grid can easily ruin long term predictions, we decided to keep our policy simple enough to avoid adding overheads in the runtime, but effective enough to fulfill our requirements of having more reliability in the sequential parts of the application or with low parallelism.

The new task replication mechanism can work together with the existing ones. We treat replicas as regular tasks, and in case they have a failure or must be cancelled, they may be resubmitted to a new resource. This way, replicas are able to overcome temporary failures and they help in detecting malfunctions in machines which belong to our Grid. Soft and hard timeouts are different in different machines for the same task, as they are calculated depending on the computational power of the resources. Also, the runtime must control the working disks of the replicas to decide where to replicate a task to avoid concurrency problems (two tasks writing in the same result files).

In brief, the implemented algorithm is described as follows. It will be executed in every scheduling step, after submitting all ready tasks to the available machines in the Grid. When no more ready tasks are available, we can use the remaining free machines to execute replicas of the running tasks. For all the running tasks, we check if the task has been already replicated (or tried to replicate unsuccessfully) and if the replication threshold is reached. The replication threshold is defined as the minimum execution time needed to consider the task for replication. Delaying the replication of a task is useful because the data dependency graph is created and consumed at the same time, dynamically and from the start of the application. So, when the application starts, we cannot directly consider replications until an initial set of tasks has been generated. Then also prevents the replication of tasks that are going to generate directly an application related error which comes from the implementation of the task (memory leaks, division by 0, ...).

The selection of the most suitable machine to execute the task replica is done the same way as with a regular task: estimating the execution time, the file transfer time (considering locality), and selecting the smallest estimation (see Section 4.2.3). However, in the resource selection, we do not consider the machine where the original task is already running as a candidate to submit a replica. One of the objectives of task replication is to avoid possible failures in machines when executing a task, then it makes no sense to replicate a task in the same machine where it is already running. In addition, if we replicate a task in the same machine we will not be able to speed up the execution, as we cannot expect the replica to be executed faster.

When one of the replicated task finishes, the runtime will submit a cancel to the rest of running replicas (1 in current implementation) and they will be marked as discarded. So, when the runtime receives a notification about a successful cancellation or a *done* status in a replica, it will mark the possible results which the task may have generated in the remote machine as files to be erased in the final post-process of the execution.

In summary, the main difference of our system in comparison with others is that replicas are defined and scheduled dynamically, on availability of the Grid resources and on demand of the application. Some of these replicas will be added to increase the probability of task completion and also will speed up the execution in case of Grids with a high level of heterogeneity (very fast nodes against very slow ones). Besides, retries will be always defined and executed under failure, which will ensure the completion of the application. Our replication mechanism is also very useful for applications with a variable degree of parallelism: when no tasks can be submitted in parallel, we replicate tasks into free resources to increase their probability of being executed correctly or to speed up their execution. Thus, sequential parts of the application, which are critical to determine the total execution time, are replicated to avoid delaying the whole execution if a failure appears.

5.3.2 Experimental results

To test our dynamic task replication mechanism we have used again the fastDNAml algorithm already presented in Section 4.3.2. This algorithm is suitable to test this mechanism, as in the data dependence graph there are synchronization points which are very important for the performance achieved by the application.

The Grid for this experiment is formed with a total number of 60 CPUs, with machines from a single PC to a cluster of 29 CPUs. They are distributed in three different administrative domains around Spain (Ciudad Real, Madrid and Barcelona). These computers have Globus installed, as well as the latest GRIDSs distribution. It is important to mention that this Grid is an heterogeneous environment, with slow and fast machines mixed.

We have run the fastDNAml algorithm with the same input used in the HPC Challenge in SC2003, where an MPI version of the algorithm won the award to the *Most Distributed Application* [93]. We have tested all the clustering policies defined in a first set of runs without task replication, and in a second set with this new mechanism activated in the runtime. Table 5.2 contains the total execution time of the different runs, detailing if the replication mechanism is enabled and the policy used to pack the trees. We have defined the 12 first iterations to be executed in the master, and packages of 4 trees when the variable MAX_PACKET_SIZE applies (UNFAIR and DEFAULT policies).

The first thing we notice is the big penalty paid with the FAIR policy, which may be suitable for homogeneous Grids, but it is clearly not for heterogeneous. Comparing

Table 5.2 Time elapsed for execution

Policy	No Replication	Replication	Reduction
FAIR	36213 s	25572 s	-29.4%
UNFAIR	21534 s	19182 s	-10.9%
DEFAULT	21497 s	19040 s	-11.4%

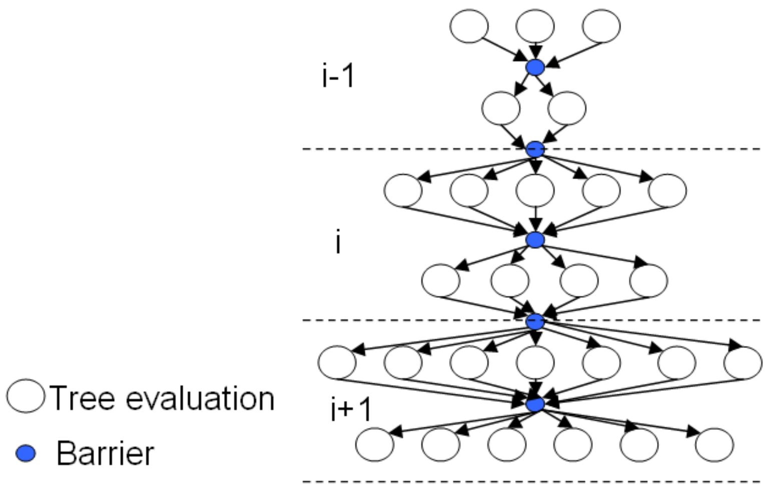


Figure 5.9 fastDNAml workflow generated by GRIDSs

the execution times we see a gain in the total execution time for all the policies. This is explained with the failures caused during the execution, the data dependency structure of the algorithm and with the heterogeneity of the Grid configuration. The evaluations in the algorithm generate a task dependency graph such as the one shown in Figure 5.9. In every evaluation step, all tasks can be submitted in parallel to our Grid, and faster machines will execute more tasks than the slower ones. After the evaluation step, there is a synchronization point to decide which of the reordered trees is the best (has the highest overall likelihood value).

In the FAIR policy there is an undesired side effect which is produced by the slowest machine in the pool. As work is divided among all available machines, slow CPUs get the same amount of work as faster ones. Without replication, we must wait for the slowest machine in every synchronization step. With replication, when a fast machine finishes its work, it will receive a replica of tasks still running, similar to what would be done with opportunistic migration, thus speeding up the execution.

The situation is different in the DEFAULT policy, because in every iteration we have free slots to replicate tasks, but this replicas go to slowest machines, which have not been initially selected to run jobs. We emulated some failures in replicated tasks, which caused a resubmit of the task in the machine where it was running. However, the other copy of the task kept running, so the penalty paid for every failure has been a slower execution time in case we killed the task in the faster machine. When we killed the task in the slower one, no penalty is paid. Without replication, we always would pay the penalty of resubmitting the task in the machine from the start. The UNFAIR policy behaves exactly as the DEFAULT policy because there are less tasks than CPUs in every iteration step.

In conclusion, the replication mechanism is useful, not only to avoid delaying the total execution time when failures appear in an application, but also to speed up the total execution time by replicating tasks running in slower machines to faster ones, without demanding users to implement a checkpoint mechanism for their tasks. This also may be applied to many other algorithms with parallel and sequential parts during their execution: failures in parallel parts are covered with retries, and failures in sequential parts are shielded with task replication to ensure they are not delayed because a failure arises.

Chapter 6

Conclusions

In this thesis we have studied the current state of the art in programming models for the Grid. From the previous work done in this area, we can see the importance of this topic for the research community, as many groups have dedicated their efforts in easing the programming of Grid applications, but following very different directions. As we have seen in Chapter 2, different systems try to provide solutions to avoid that users have to program directly the Grid services, which is a very difficult task for them. The ideal scenario is that a user knows the minimum about the Grid technology underneath their application. This approach is known as *Grid-unaware* and it is one of our main objectives in our research: to keep the Grid as invisible as possible to the end user.

6.1 Programming interface

Our contribution to that objective is GRID superscalar, a programming model which helps in the development of applications to be run in Grid environments and exploits the existent parallelism of the applications at the program level. The first part of the thesis presents the GRIDSs interface. The interface has been designed to be very simple. A user must provide three basic components: the main program of the application, the code of the functions that they want to run in the Grid and an IDL file describing the interface of those functions. The main program and the functions can be programmed in different languages: C, C++, Java, Perl and Shell Script. With this, GRIDSs acts as an intermediate language, as the assembler language does between programming languages and the machine language. The set of calls defined to be used in the main program and functions is very small to keep the programming interface very simple to learn for end users. There are 8 primitives available in the master side (4 of them related to open and close files), and 2 in the worker side.

Some extra primitives to achieve the speculative execution of tasks are provided. Having a group of speculative tasks means that, when a condition is reached, the tasks after the one that reached that condition will not be executed and can be discarded. This has been designed specifically for parameter-sweep applications or optimization algorithms, that typically submit a big set of jobs in order to run the same test with different inputs, and when an objective value is reached, the rest of the tasks are not required to be executed. With this mechanism, the parallel execution of this type of applications can be exploited, but also the computation will not be more than needed.

The interface also includes the possibility of defining the constraints and the computation cost of a task in order to execute it. The constraints are very useful when the Grid where an application must be run is heterogeneous: with machines from clusters to regular PCs, different operating systems, installed software, physical memory, architecture, ... With the constraints specification a user can specify what is needed in order to run a task, and the runtime checks what machines of the total available accomplish the requirements to run it. The cost can also be expressed by users, based in previous runs of the operations they want to execute in the Grid. This cost can be described as a formula to calculate how long the task will take in a machine, using the size of the input files and the computational power available in the machine where GRIDSs is considering to submit a job.

We also provide a comparison between our programming model and other existing environments to prove that it is very simple and easy to use to program an application for the Grid. We compare our model with programming directly a Grid middleware (Globus), using a workflow tool (DAGMan), using a programming model based on the GridRPC API (Ninf-G), and using a language to build workflows (VDL). In these cases, our model has a simpler interface because the parallelism and the Grid details are not specified in the code. Moreover, as it is based in an imperative language, it allows to build big workflows in an easier way than with other mentioned tools.

6.2 Runtime

The runtime of GRIDSs is the one in charge of generating tasks for every function the user wants to execute in the Grid. Every call to a function generates a task in the runtime, which builds automatically a Direct Acyclic Graph describing the data dependencies between the tasks. All tasks that do not have preceding data dependencies are ready to run, and if there are resources available, we can submit them to run concurrently in the Grid. The data dependencies are analyzed taking into account the files every task creates or consumes. This is justified with the kind of tasks that can benefit from using the Grid: tasks with high computation demands (bioinformatic applications, chemistry simulations, ...). These applications usually work with large inputs that are stored in files. The data dependencies detected are the same as in a superscalar processor [47]: RaW, WaR and WaW. In order to increase the parallelism available in the dependency graph, we eliminate WaR and WaW dependencies by means of a renaming technique. File names are renamed internally and transparently for the end user.

Although the scheduling of tasks was not one of our main research objectives, we have done some work in this topic. In order to submit a task, the task dependency graph is the basis which defines what tasks can be submitted to the Grid because they do not have any data dependencies pending to be solved. When a task finishes its execution, the data dependency graph is updated by erasing the finished task, which may release other tasks in order to be executed if they become free of preceding dependencies.

Regarding resource brokering, we have developed a simple broker which controls the Grid machines (what machines can be used and their configuration). The scheduling policy in the broker is focused to reduce the total execution time of a task. To achieve this, it uses a greedy policy which selects the task with the smallest total execution time estimation from the ones available to be run. The runtime considers not only the time the task will take in order to be run (provided by the user), but also the time required to transfer the input files needed for the task. The estimation of the execution time of a task is machine dependant and dynamically calculated for every task. In addition, task constraints and machine capabilities are matched. This is implemented using the ClassAds [77] library in the runtime. This library allows to publish "advertisements" where machines offer capabilities and tasks require them.

Our system includes a file locality exploitation policy to reduce the transfers between different resources in the Grid to the minimum, thus saving time and network usage during an execution. When a task finishes its execution in a remote machine, its output files are kept there in order to save future transfers: if a new task needs those results as inputs, submitting it to this machine means that we will not need to transfer them. So, when estimating the time a task will spend in a machine to run, we will not count the time to transfer its inputs if they are already available there. Our environment maintains information of the location of all files involved in the computation, even if they are inputs sent, or outputs generated after a task execution.

Related to the locality of files, other features have been explained, as the possibility of defining shared disks among machines. This is very important because in a distributed system the usage of replicas or shared spaces is very common in order to access to some data faster, or access it transparently regardless its location. GRIDSs allows the definition of machines that share a working space, or also replicated input directories in different machines. Defining those zones lets the runtime know when a file transfer will not be needed. Another described feature is the deletion of intermediate versions of a file. As the renaming mechanism generates several versions of a single file, when there is a new name for a file future references to that file will use the new name, so, previous versions are candidates to be erased. When an old version is no more needed it will be erased from all the machines where it is available, reducing the total disk usage. Related both to the locality and the constraints specification, there is also a quota control implemented, in order to know in advance if the task will have enough space for its files to run.

Some information needs to be sent to the master machine in order to determine if the task has finished correctly, to send output scalars and to gather some information needed for the scheduler. We have presented two mechanisms available to get this information: the file mechanism and the socket mechanism. While the former is suitable when machines have a limited connectivity (firewalls, ...), the latter achieves a faster communication when the machine does not have such connectivity restrictions.

We have presented several evaluation tests of the runtime which pursue very different objectives. We have made a deep analysis of the runtime when executing the NAS Grid Benchmarks with GRIDSs and analyzing some of the executions in detail. Moreover, we have studied the performance of a simple optimization example and several real use cases of the tool, demonstrating that not only the data dependency graph but also the granularity of tasks have an important influence in the final speed up achieved by the application when run in the Grid. Other real use cases show our experience in using a transoceanic Grid and a big cluster with 1000 CPUs, which is a high demanding environment for our runtime. Finally, a case study comparing PACX-MPI and GRIDSs is also presented.

6.3 Fault tolerance mechanisms

The last part of the thesis presents the study to include fault tolerant mechanisms in our environment, which are extremely important in a distributed environment such as the Grid (prone to failures). Failures can be related to the application (memory leaks, ...), related to the underlying Grid services (job submission, file transfer, ...) or related to the Grid resources (machine crash, network failure, ...). The mechanisms studied that have been implemented in the runtime in order to deal with failures are checkpoint, retry mechanisms and task replication. With this addition, GRIDSs is able to deal with errors at all levels, providing a complete and easy solution for end users.

The checkpoint is a mechanism focused to save the work previously done in case of an unrecoverable failure. Thus, the user is able to restart the application where it was stopped due to that failure instead of having to start the application from the beginning. The checkpoint in GRIDSs works at a inter-task level. This means that it will save the state of tasks that have completed their execution. Tasks that have not finished will be restarted from the beginning. In our strategy to checkpoint tasks, we assume we can checkpoint a task when all tasks with a smaller sequence number have finished correctly. The sequence number is established by the sequential order of the application (the way it would have been executed sequentially in a single machine). This assumption simplifies the checkpoint complexity, avoiding to store all data structures from memory to disk and reducing the number of files that we must maintain in order to restart the application correctly. We say that this mechanism recovers the sequential consistency of the application. We have seen that the drawback of having to discard some finished tasks can be reduced by influencing the scheduling decisions. However, this mechanism is completely transparent to end users, and they do not need to provide anything special in order to use the checkpoint. A simple optimization example is presented in order to prove that our checkpoint mechanism does not add overhead to the application run time when no failures are detected.

We have proven that the retry mechanisms included are a good way to overcome failures at run time allowing the application to continue its execution. More precisely, the mechanisms proposed include the possibility of dropping a machine from the computation, soft and hard timeouts for tasks, asynchronous transfer of task results and to retry Grid middleware operations and system calls in the runtime.

When there is a certain amount of failures in a machine, it is interesting to remove it from the computation to avoid future failures. Anyway, the dropped machine can be added again to the computation when the user fixes its problems. Tasks have two timeouts defined in order to detect possible failures. The first one is called soft timeout and it is defined as the time when the runtime will start asking for the status of that task to the Grid services. The second timeout is the hard timeout and it is described as the maximum amount of time a task will be running. This mechanism allows to consider severe performance degradations as failures and act to correct them. The asynchronous transfer of results in the runtime avoids the need to undo checkpoints if the system is not able to access result files stored in a remote machine. This increases the availability of these files and reduces the post-process time of GRIDSs. Another interesting feature derived from this mechanism is the possibility of analyzing results of the execution in the master while the application is running. Retry mechanisms are also applied in case of a

Grid service request failure and system call failures with the objective that the execution is only stopped when an unrecoverable failure appears. We provide an test case with a long run Monte Carlo algorithm to see how our mechanisms deal with a variety of Grid errors.

A replication mechanism has also been studied in GRIDSs. It allows not only to deal with Grid level errors but also to increase the performance in heterogeneous environments without the need of an opportunistic migration mechanism. The migration mechanism provided in most systems requires an implementation of a checkpoint strategy from the user, which we do not demand. The replication is done based on the free machines available while executing the application and on the task dependency graph that GRIDSs has generated from the application. Thus, sequential parts of the application or parts with few parallelism (giving priority to tasks that have more successors) are replicated to ensure that their data dependencies will be solved even in case of a failure. Only free machines are considered to run replicas in order to avoid overloading the Grid resources. We have used the fastDNAml algorithm to demonstrate that the replication mechanism is useful not only to avoid delaying the total execution time when failures appear in an application, but also to speed up the total execution time by replicating tasks running in slower machines to faster ones, without demanding users to implement a checkpoint mechanism for their tasks.

With all the fault tolerance features added to our system, GRIDSs becomes an integrated programming environment that includes failure detection and recovery. Our system offers a complete fault tolerance solution able to deal with all types of failures in complex workflow applications easily. Application numerical errors can be handled when coding the application. The impact of application unrecoverable failures is reduced with the checkpoint, which is transparent for users. Grid errors are also handled transparently to users with the retry mechanism, getting the run time estimation of a task as the information source to: determine the best moment to ask about the status of the task to avoid introducing overhead, and detect a severe performance degradation in the task's execution.

In brief, this thesis document describes our contributions to the research community. We have created a programming model that has helped users to run their applications in the Grid in an easy manner, with a powerful runtime that allows to execute those applications efficiently in a Grid. We have added fault tolerant features to the system, which are very important in a Grid distributed system, where failures can happen with a higher probability compared to other environments. We have presented experimental results and successful stories where GRIDSs is used. Besides, GRIDSs has been also *a source of inspiration in some European Union Integrated Projects (IP) and Network of Excellence: CoreGRID, BEinGRID, BREIN, XtreemOS. Also in LAGrid project and the Spanish initiative* Red Temática para la Coordinación de Actividades Middleware en Grid.

6.4 Future work

The main goal of GRIDSs is to assist users in exploiting the power of the Grid in their applications. Thus, one of the main objectives that we always have present is to meet new users to show them the benefits of using the Grid technology. We have seen how applications from different fields can benefit from our programming model, so, many other similar applications would obtain the same benefit when using GRIDSs. Our future work always includes to make more popular our system and test new applications from different scientific fields, demonstrating the simplicity of using our programming interface and how parallelism can speed up their algorithms. One big help in this sense has been the MareNostrum supercomputer [16]. It has been mentioned in this document that there is a version of GRIDSs that works with ssh and scp between the nodes of MareNostrum [9]. This means that MareNostrum users are able to use GRIDSs to program their applications and thus avoid the need of implementing scripts to submit and manage their jobs to the queuing system in the supercomputer. Moreover, we are currently extending this MareNostrum solution to the recently created Spanish Supercomputing Network *(Red Española de Supercomputación)*. This network has been formed with parts of the first version of the MareNostrum supercomputer distributed between different institutions in Spain. GRIDSs will work then in a Grid of supercomputers, and will allow the users to run their experiments using several supercomputers at the same time transparently for them.

In this thesis document we have seen experimental results, using well-known benchmarks and algorithms, such as the NAS Grid Benchmarks or the fastDNAml algorithm. These tests have been performed in small Grids in our institution, with machines from various institutions in Europe, and even with a machine located in Mexico. However, we believe it will be interesting to perform a big test by creating a big virtual organization with a high number of machines involved from around the world to see how our system performs in such an scenario. This would be a particularly good test for the fault tolerance features included in our system and would prove the robustness of GRIDSs. Also the new Spanish Supercomputing Network will create a good scenario for our tests.

During the development of GRIDSs, a simple broker was developed inside our runtime in order to deal with task submission taking into account our desired file locality policy to save transfers. However, developing a complex broker inside the runtime has never been our objective till now, considering the work that has been done in this field by other groups. So, one of the possibilities to extend our system is to integrate it with some meta-schedulers or brokers available, such as the GridBUS broker [107], the eNANOS broker [81] or GridWay [49]. These brokers should be modular enough in order to allow the implementation of our locality scheduling policy to submit jobs. A first integration between GridWay and GRIDSs has been done in the context of the BEinGRID project [10], but a deeper integration can be proposed. Another possibility is to make our broker more complex in order to add the possibility of discovering new machines in the Grid (resource discovery) and obtain resource information about the status of machines (resource monitoring). This would make our broker more dynamic and more policies related to the status of the machines could be studied. Our broker could also use the information available in the data dependency graph between tasks in order to make a more global scheduling, considering critical paths in the graph, weights in the nodes or even clustering of tasks to submit a group of tasks to the same resource.

Although it has been show that the GRIDSs interface is very simple, we are studying ways of substituting the calls to open and close files automatically in order to leave the user's source code with less changes. This would reduce the interface from 8 to 4 primitives in the master part of the application and should be done for all the programming languages supported by GRIDSs.

The presented retry mechanisms have been proven to be very useful not only to recover from temporary failures in remote machines, but also to avoid local temporary failures in the master machine by retrying system calls inside the library. However, a permanent failure in the master is still a problem, as the master controls all the retry mechanisms. We are currently studying ways to add replication in the master in order to avoid that, as it is done in the Google File System [40].

In the task replication case we have based our replication policy in the data dependency graph and the free machines available. Anyway, some interesting cases can be studied, as the degree of replication for a single task, and how many replicas should be running in the system at the same time. Related to the number of replicas, it will be interesting to study if it is better to overload the machines with replicas, even if they cause a small degradation in the execution time of other tasks. Other policies based on the execution time estimations could be done to decide which one is the most suitable task to replicate.

Some future work related to GRID superscalar is already ongoing. The GRID superscalar programming model has been the source of inspiration [8] to create a new family of programming models called Star superscalar (StarSs). The ideas presented in this thesis have been adapted to very different architectures, such as clusters (SSH GRIDSs [9]), SMP machines (SMP superscalar [74]), the Cell Broadband Engine processor (Cell superscalar [75]), and the Grid Component Model (COMP superscalar [100]). We do not discard that new versions of the programming model may be created in order to adapt it to new coming architectures.

Bibliography

[1] M. Alt, J. Dünnweber, J. Müller, and S. Gorlatch. *Component Models and Systems for Grid Applications*, chapter HOCS: Higher-Order Components for Grids, pages 157–166. Springer US, 2005. 2

[2] K. Amin, G. von Laszewski, M. Hategan, N. Zaluzec, S. Hampton, and A. Rossi. GridAnt: a client-controllable grid workflow system. *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, page 10 pp., Jan. 2004. 2

[3] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004. 2.4.2

[4] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002. 2

[5] R. Badia, V. Dialinos, J. Ejarque, J. Labarta, J. Pérez, and R. Sirvent. The GRID superscalar project: Current status. *XVII Jornadas de Paralelismo, Albacete (SPAIN)*, 2006. 1.3, 4.2.1

[6] R. Badia, R. Hood, T. Kielmann, A. Merzky, C. Morin, S. Pickles, M. Sgaravatto, P. Stodghill, N. Stone, and H. Yeom. Use-Cases and Requirements for Grid Checkpoint and Recovery. GFD.92 - Informational, Open Grid Forum (OGF), 2007. 2.4.2

[7] R. Badia, R. Sirvent, M. Bubak, W. Funika, and P. Machner. *Achievements in European Research on Grid Systems*, chapter Performance monitoring of GRID superscalar with OCM-G/G-PM: integration issues, pages 193–205. Springer, 2008. 1.3

[8] R. M. Badia, P. Bellens, M. de Palol, J. Ejarque, J. Labarta, J. M. Pérez, R. Sirvent, and E. Tejedor. GRID superscalar: from the Grid to the Cell processor. *Spanish Conference on e-Science Grid Computing, Madrid (Spain)*, 2007. 1.3, 6.4

[9] R. M. Badia, P. Bellens, J. Ejarque, J. Labarta, M. de Palol, J. M. Pérez, R. Sirvent, and E. Tejedor. SSH GRID superscalar: a Tailored Version for Clusters. *1st Iberian Grid Infrastructure Conference (IBERGRID)*, 2007. 1.3, 4.2.1, 4.2.4, 4.3.2, 6.4

[10] R. M. Badia, D. Du, E. Huedo, A. Kokossis, I. M. Llorente, R. S. Montero, M. de Palol, R. Sirvent, and C. Vázquez. Integration of GRID Superscalar and GridWay Metascheduler with the DRMAA OGF Standard. In *Euro-Par*, pages 445–455, 2008. 1.3, 6.4

[11] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003. 1.3, 4.3.1

[12] R. M. Badia, R. Sirvent, J. Labarta, and J. M. Perez. *Engineering The Grid: Status and Perspective*, chapter Programming the GRID: An Imperative Language-based Approach. American Scientific Publishers, 2006. 1.3

[13] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006. 2, 2.2.3, 3.1

[14] Z. Balaton and G. Gombás. GridLab Monitoring: Detailed Architecture Specification. Tech. Rep. GridLab-11-D11.2-01-v1.2, EU Information Society Technologies Programme (IST), 2001. 2.1.4

[15] Z. Balaton, P. Kacsuk, and N. Podhorszki. Application Monitoring in the Grid with GRM and PROVE. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 253–262, London, UK, 2001. Springer-Verlag. 2.1.4

[16] Barcelona Supercomputing Center.
http://www.bsc.es/. 6.4

[17] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96:*

Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), page 39, Washington, DC, USA, 1996. IEEE Computer Society. 2

[18] BLAST Home Page.
http://www.ncbi.nlm.nih.gov/BLAST/. 3.1.1, 4.3.2

[19] R. Brobst, W. Chan, F. Ferstl, J. Gardiner, J. P. Robarts, A. Haas, B. Nitzberg, H. Rajic, and J. Tollefsrud. Distributed Resource Management Application API Specification 1.0. GFD.022 - Full Recommendation, Open Grid Forum (OGF), 2007. 4.2.1

[20] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, 2003. 2.4.2

[21] Business Experiments in Grid.
http://www.beingrid.eu/. 4.3.2

[22] The Home of Checkpointing Packages.
http://checkpointing.org/. 5.1.1

[23] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. *Supercomputing, ACM/IEEE 2002 Conference*, page 58, 16-22 Nov. 2002. 2.1.2, 4.2.4

[24] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth Int. Conference on*, pages 356–363, 1999. 5.3.1

[25] J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004. 1.1

[26] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid Information Services for Distributed Resource Sharing. *hpdc*, 00:0181, 2001. 2.1.2

[27] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping Abstract

Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39, 2003. 2

[28] R. V. der Wijngaart and M. Frumkin. NAS Grid Benchmarks: A Tool for Grid Space Exploration. In *in Proc. of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10âĂŹ01)*, pages 315–324, August 2001. 4.3.1

[29] V. Dialinos, R. Badia, R. Sirvent, J. Perez, and J. Labarta. Implementing phylogenetic inference with GRID superscalar. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2005)*, 2:1093–1100, 9-12 May 2005. 1.3, 4.3.2, 4.3.2

[30] Directed Acyclic Graph Manager Homepage (DAGMan). http://www.cs.wisc.edu/condor/dagman/. 2, 2.1.1, 2.1.2, 3.3.2

[31] D. W. Erwin and D. F. Snelling. *Euro-Par 2001 Parallel Processing*, chapter UNICORE: A Grid Computing Environment, pages 825–834. Springer Berlin / Heidelberg, 2001. 1.1, 3.1

[32] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: a Grid application development and computing environment. *The 6th IEEE/ACM International Workshop on Grid Computing*, 13-14 Nov. 2005. 2.4.2

[33] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997. 1.1, 2.1.4, 2.2.1, 3.1, 4, 4.2.1, 5.2.1

[34] I. Foster and C. Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 1.1

[35] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. *Proceedings 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, 2002. 2, 2.1.2, 3.3.4

[36] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), Nov. 1996. Updated by RFCs 2184, 2231, 5335. 4.2.1

[37] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 55–63, San Francisco, California, August 2001. 1.1, 2.1.1

[38] The General Atomic and Molecular Electronic Structure System. http://www.msg.ameslab.gov/GAMESS/. 3.1.1, 4.3.2

[39] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994. 2.1.4

[40] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles*, pages 29–43, 2003. 6.4

[41] A. Gianelle, R. Peluso, and M. Sgaravatto. Datagrid: Job Partitioning and Checkpointing. Technical report, CERN, 2002. 2.4.2

[42] T. Goodale, G. Allen, G. Lanfermann, J. Massó, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *Proceedings of the 5th International Conference in Vector and Parallel Processing - VECPAR'2002*. Springer, 2003. 2.4.2

[43] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). GFD.090 - Proposed Recommendation, Open Grid Forum (OGF), 2008. 2, 4.2.1

[44] The Grid Programming Environment. http://sourceforge.net/projects/gpe4gtk/. 2

[45] Grid Remote Procedure Call Working Group. http://www.ogf.org/gf/group_info/view.php?group=gridrpc-wg. 2

[46] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. *21th Symp. on Reliable Distributed Systems (SRDS)*, 2002. 2.4.1

[47] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002. 1.3, 1.3, 4.1.1, 4.1.2, 6.2

[48] A. Hoheisel. User Tools and Languages for Graph-based Grid Workflows. *Grid Workflow Workshop, GGF10*, 2004. 2

[49] E. Huedo, R. S. Montero, and I. M. Llorente. A framework for adaptive execution in grids. *Software: Practice and Experience*, 34(7):631–651, 2004. 2.4.1, 4.2.3, 4.2.4, 4.3.2, 6.4

[50] S. Hwang and C. Kesselman. Grid workflow: A flexible failure handling framework for the Grid. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 126–137, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 2.4.2

[51] OMG IDL Syntax and Semantics.
http://www.omg.org/cgi-bin/doc?formal/02-06-39/. 3.2.2

[52] M. Juan and R. M. Badia. Ampliació dels llenguatges suportats per l'entorn GRID superscalar. Graduate Thesis, Universitat Politècnica de Catalunya, 2005. 3.1.3

[53] P. Kacsuk, G. Dozsa, J. Kovacs, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombas. P-GRADE: A Grid Programming Environment. *Journal of Grid Computing*, 1(2):171–197, 2003. 2.1.4, 2.4.2

[54] W. Kang, H. H. Huang, and A. Grimshaw. A Highly Available Job Execution Service in Computational Service Market. In *Proceedings of the 8th IEEE/ACM International Grid Computing Conference*, 2007. 2.4.2

[55] S. Kannan, P. Mayes, M. Roberts, D. Brelsford, and J. F. Skovira. *Workload Management with LoadLeveler*. IBM Redbooks, 2001. 2.4.2

[56] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003. 2, 4.3.2

[57] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004. 2.4.1

[58] E. Krepska, T. Kielmann, R. Sirvent, and R. M. Badia. *Achievements in European Research on Grid Systems*, chapter A Service for Reliable Execution of Grid Applications, pages 179–192. Springer, 2008. 1.3, 2.4.2

[59] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. *Euro-Par'96 Parallel Processing*, chapter DiP: A parallel program development environment, pages 665–674. Springer Berlin / Heidelberg, 1996. 3.2.1, 4.3.1, 4.3.1, 4.3.2

[60] E. Laure, S. M. Fisher, A. Frohner, C. Grandi, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, F. Hemmer, A. D. Meglio, and A. Edlund. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006. 3.1

[61] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou. Efficient task replication and management for adaptive fault tolerance in Mobile Grid environments. *Future Generation Computer Systems*, 2007. 2.4.2

[62] R. Lovas, R. Sirvent, G. Sipos, J. M. Pérez, R. M. Badia, and P. Kacsuk. *Integrated Research in GRID Computing*, chapter GRID superscalar enabled P-GRADE portal, pages 241–254. Springer, 2007. 1.3

[63] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, May 1999. 2.2.2

[64] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in grids: Why are they so bad and what can be done about it? In *Proceedings of the Fourth International Workshop on Grid Computing*, pages 18–24, 2003. 2.4.2

[65] L. Moreau, Y. Zhao, I. Foster, J. Voeckler, and M. Wilde. XDTM: The XML Dataset Typing and Mapping for Specifying Datasets. In *Proceedings of the 2005 European Grid Conference (EGC'05)*, Amsterdam, The Netherlands, 2005. 2.2.5

[66] The Message Passing Interface (MPI) standard.
http://www-unix.mcs.anl.gov/mpi/. 1.1, 2.1.4, 3.1.1

[67] M. O. Neary, A. Phipps, S. Richman, and P. R. Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1231–1238, London, UK, 2000. Springer-Verlag. 2

[68] T.-A. Nguyen and P. Kuonen. ParoC++: A Requirement-Driven Parallel Object-Oriented Programming Language. In *IPDPS '03: Proceedings of the 17th*

International Symposium on Parallel and Distributed Processing, page 129.1, Washington, DC, USA, 2003. IEEE Computer Society. 2

[69] G. J. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek. fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Comput. Appl. Biosci.*, 10(1):41–48, 1994. 4.3.2

[70] The Open Grid Forum (OGF).
http://www.ogf.org/. 2.4.2

[71] The OpenMP Homepage.
http://www.openmp.org/drupal/. 1.1, 4.2.3

[72] The PACX-MPI Project Homepage.
http://www.hlrs.de/organization/amt/projects/pacx-mpi/. 2, 4.3.2

[73] J. M. Pérez and R. M. Badia. Millora de l'entorn de programació GRID superscalar. Graduate Thesis, Universitat Politècnica de Catalunya, 2004. 3.1.2

[74] J. M. Perez, R. M. Badia, and J. Labarta. A flexible and portable programming model for SMP and multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center, June 2007. 6.4

[75] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Programming the Cell/B.E. made easier. *IBM Journal of R&D*, 51(5), August 2007. 6.4

[76] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: Fast and Light-weight tasK executiON framework. In *Proceedings of the Supercomputing Conference (SC'07)*, 2007. 2.2.5

[77] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998. 3.2.5, 4.2.3, 6.2

[78] The RealityGrid project.
http://www.realitygrid.org/. 2.4.2

[79] S. Reyes, C. Munoz-Caro, A. Nino, R. M. Badia, and J. M. Cela. Performance of computationally intensive parameter sweep applications on internet-based grids of computers: the mapping of molecular potential energy hypersurfaces: Research

articles. *Concurrency and Computation: Practice and Experience*, 19(4):463–481, 2007. 4.3.2

[80] S. Reyes, A. Nino, C. Munoz-Caro, R. Sirvent, and R. M. Badia. Monitoring Large Sets of Jobs in Internet-Based Grids of Computers. *1st Iberian Grid Infrastructure Conference (IBERGRID)*, 2007. 1.3

[81] I. Rodero, J. Corbalán, R. M. Badia, and J. Labarta. *Advances in Grid Computing - EGC 2005*, chapter eNANOS Grid Resource Broker, pages 111–121. Springer Berlin / Heidelberg, 2005. 4.2.3, 6.4

[82] E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. GridLab: a grid application toolkit and testbed. *Future Gener. Comput. Syst.*, 18(8):1143–1153, 2002. 2.4.2

[83] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, 18 Nov. 2002. 2, 2.2.1, 3.3.3

[84] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530 (Proposed Standard), Apr. 2003. 4.2.4

[85] Simple API for Grid Applications Working Group.
http://www.ogf.org/gf/group_info/view.php?group=saga-wg. 2

[86] R. Sirvent, R. M. Badia, P. Bellens, V. Dialinos, J. Labarta, and J. M. Pérez. Demostración de uso de GRID superscalar. *Boletín RedIRIS*, (no. 80):41–46, Abril 2007. 1.3

[87] R. Sirvent, R. M. Badia, N. Currle-Linde, and M. Resch. *Achievements in European Research on Grid Systems*, chapter GRID Superscalar and GriCoL: Integrating Different Programming Approaches, pages 139–150. Springer, 2008. 1.3

[88] R. Sirvent, R. M. Badia, and J. Labarta. Fault tolerance and execution speed up with task replication for scientific workflows. *High Performance Computing and Communications (HPCC) (Submitted)*, 2009. 1.3

[89] R. Sirvent, A. Merzky, R. M. Badia, and T. Kielmann. GRID superscalar and SAGA: forming a high-level and platform-independent Grid programming environment. *CoreGRID Integration Workshop. Integrated Research in Grid Computing*, 2005. 1.3

[90] R. Sirvent, J. M. Pérez, R. M. Badia, and J. Labarta. Automatic Grid workflow based on imperative programming languages: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1169–1186, 2006. 1.3

[91] R. Sirvent, J. M. Pérez, R. M. Badia, and J. Labarta. Programación en el Grid: una aproximación basada en lenguajes imperativos. *Computación Grid: del Middleware a las Aplicaciones*, pages 33–52, Junio 2007. 1.3

[92] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999. 2.4.1

[93] C. Stewart, D. Hart, M. Aumuller, R. Keller, M. Muller, H. Li, R. Repasky, R. Sheppard, D. Berry, M. Hess, U. Wossner, and J. Colbourne. A global grid for analysis of arthropod evolution. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 328–337, 8 Nov. 2004. 4.3.2, 5.3.2

[94] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, and C. Vizino. A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System. Technical report, Pittsburgh Supercomputing Center, 2003. 2.4.2

[95] N. Stone, D. Simmel, T. Kielmann, and A. Merzky. An Architecture for Grid Checkpoint and Recovery Services. GFD.93 - Informational, Open Grid Forum (OGF), 2007. 2.4.2

[96] H. Takemiya. Constructing Grid Applications on the World-Wide Testbed - Climate Simulation and Molecular Simulation. *6th HLRS Metacomputing and GRID Workshop*, May 2003. 4.3.1

[97] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003. 2, 2.2.1, 3.3.3, 4.2.1

[98] Taverna Project Website.
http://taverna.sourceforge.net/. 2

[99] I. Taylor, M. Shields, and I. Wang. Distributed P2P computing within Triana: A Galaxy Visualization Test Case. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8 pp.–, 22-26 April 2003. 2, 2.1.3, 3.1

[100] E. Tejedor and R. M. Badia. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 185–193, Washington, DC, USA, 2008. IEEE Computer Society. 6.4

[101] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. 2.2.1, 2.4.2

[102] S. Thatte, T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1, 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/. 2, 2.1.3

[103] The Unified Modeling Language Web Page.
http://www.uml.org/. 4.2.1

[104] R. V. van Nieuwpoort, T. Kielmann, and H. Bal. User-Friendly and Reliable Grid Computing Based on Imperfect Middleware. In *Proceedings of SC07*, 2007. 2.4.1

[105] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005. 2, 2.2.2

[106] S. Venkatakrishnan, M. Kuchhal, and S. Kumar. Grid Application Framework for Java (GAF4J). http://alphaworks.ibm.com/tech/GAF4J/. 2

[107] S. Venugopal, R. Buyya, and L. J. Winton. A Grid service broker for scheduling e-Science applications on global data Grids. *Concurrency and Computation: Practice and Experience*, 18(6):685–699, May 2006. 2.4.1, 4.2.3, 6.4

[108] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001. 2.2.1, 2.2.4

[109] G. von Laszewski, M. Hategan, and D. Kodeboyina. *Workflows for e-Science*, chapter Java CoG Kit Workflow, pages 340–356. Springer London, 2007. 2.2.4

[110] Web Services Choreography Working Group.
http://www.w3.org/2002/ws/chor/. 2

[111] J. B. Weissman. Fault tolerant computing on the grid: what are my options? In *Proceedings of The Eighth International Symposium on High Performance Distributed Computing*, pages 351–352, 3-6 Aug. 1999. 2.4.2

[112] G. Wrzesinska, R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1):103–114, Spring 2006. 2.4.2

[113] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Record*, 34(3), 3 Sept. 2005. 2.4.2

[114] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. *IEEE International Workshop on Scientific Workflows*, 2007. 2, 2.2.5, 3.1

Appendix A

Runtime environment variables

As the requirements of users and applications may differ when using GRID superscalar, we provide a set of environment variables which can be defined before running an application. This allows an easy tuning of the runtime behavior that can be done by expert users without the need of recompiling neither the GRID superscalar library nor the application's code. This variables have a default value that is only modified when users specify a new value in their environment, overwriting the default value. The list of variables is:

- GS_DEBUG: this variable can be set to receive more or less debug information during the execution. When its value is 20, the master will write at its standard output many debug information to allow users to determine potential problems during the execution. When set to 10, less information will be printed with the objective of following the execution of the tasks. Default value is 0, which means no debug information is requested.

- GS_LOGS: a value of 1 tells the master to leave execution logs of all tasks executed in all worker machines. These logs will be named OutTaskXX.log and ErrTaskXX.log, according to the standard output and standard error messages given in that task (where XX is the number of the task, defined by the sequential order of the application. Default value is 0, indicating that no logs should be provided.

- GS_SOCKETS: related with the end of task message mechanisms implemented in the runtime. If it is set to 1, messages from workers to the master will be sent through sockets. Default value is 0 to enable the communication through files.

- GS_MIN_PORT: this variable only applies when working with GS_SOCKETS set to 1. Some machines have constraints in connectivity, regarding to opened ports.

For this reason an available range of ports may be indicated to GRIDSs to be used to open a reply port when working with socket notifications.

- GS_MAX_PORT: the upper threshold of the range of ports. It is only considered when GS_SOCKETS is set to 1.

- GS_ENVIRONMENT: this variable is considered an advanced feature. Some extra environment variables could be needed to be passed when executing your jobs in the Grid (i.e. when your jobs are parallel). These variables can be passed with this parameter. Your GS_ENVIRONMENT string can be as long as pointed by GS_MAXPATH. Each variable must be in parentheses: (VARIABLE1 value1)(VARIABLE2 value2), ... The content of GS_ENVIRONMENT will be sent to each worker machine.

- GS_TIMEOUT_FACTOR: this factor is multiplied by the estimated execution time given by the user for a task, and determines a soft limit for the real execution time of the task. Once this limit is reached, the runtime starts to ask if the job is still running or there has been an error. This belongs to the fault tolerance mechanisms included in the runtime, which are explained in Chapter 5. Default value is 1.25 and it must be bigger than 1 when redefined. If it is set to 0 it will disable the resubmit mechanism.

- GS_RESUBMIT_FACTOR: the resubmit factor is also multiplied by the estimated execution time for a task, determining a maximum execution time for that task. So, if this time is reached, the task is killed and resubmitted. This mechanism is also explained in Chapter 5. Default value is 2 and it must be bigger than the specified timeout factor.