UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ELTON NICOLETTI MATHIAS

Hierarchical Message Passing through a ProActive/GCM based Runtime

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Nicolas Maillard
Advisor

Prof. Françoise Baude
Coadvisor

Porto Alegre, March 2010

CIP – CATALOGING-IN-PUBLICATION

Mathias, Elton Nicoletti

Hierarchical Message Passing through a ProActive/GCM based Runtime / Elton Nicoletti Mathias. – Porto Alegre: PPGC da UFRGS, 2010.

116 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Advisor: Nicolas Maillard; Coadvisor: Françoise Baude.

1. Parallel programming. 2. Component-oriented programming. 3. Programming model. 4. Grid programming. 5. Messsage passing. 6. MPI. I. Maillard, Nicolas. II. Baude, Françoise. III. Título.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ADI	Abstract Device Interface
API	Application Programming Interface
CCA	Common Component Architecture
CCM	CORBA Component Model
CEM	Computational Electromagnetism
CFD	Computational Fluid Dynamics
CORBA	Common Object Request Broker Architecture
FIFO	First In First Out
GCM	Grid Component Model
HOC	High-Order Components
HTTP	HyperText Transfer Protocol
I/O	Input/Output
INRIA	Institut National de Recherche en Informatique et en Automatique
JVM	Java Virtual Machine
LAN	Local Area Network
LRU	Least Recently Used
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NUMA	Non-uniform Memory Access
OGSA	Open Grid Services Architecture
OS	Operating System
P3D	Poisson3D (the application)
PDE	Partial Differential Equations
RMA	Remote Memory Access
RMI	Remote Method Invocation
RPC	Remote Procedure Call

SMP	Symmetric Multiprocessing
SPMD	Single Process, Multiple Data
TCP/IP	Transmission Control Protocol / Internet Protocol
URL	Uniform Resource Locator
VN	Virtual Node
VO	Virtual Organization
WAN	Wide Area Network
XML	Extensible Markup Language
XSD	XML Schema Definition

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the past several years, grid computing has emerged as a way to harness computing resources geographically distributed across multiple organizations. Due to its inherently largely distributed and heterogeneous nature, grid computing has enlarged the importance of specific requirements, such as scalability, performance and the need of an adequate programming model. Several programming models have been proposed for grid programming. Nonetheless, so far, none of them met all the requirements. Differently, in the field of high performance cluster computing, the message passing model became a true standard with a large number of libraries and legacy applications.

This work proposes a hybrid framework that combines the high performance and high acceptability of the MPI standard boosted with intuitive extensions to enable developers to design grid applications or *"gridify"* existing ones with the flexibility of a component-based runtime modeling resources hierarchy and offering support to inter-cluster communication. The proposed solution relies on the addition of new MPI communicators and a related API, which may offer a support well-suited to programmers used to MPI in order to reflect a hierarchical topology within the deployed application.

The experiments with some applications with different characteristics (Monte-Carlo Simulation, a Mergesort and a Poissond3D solver) have shown that the *"gridification"* of applications improve application performance on grid environments. Even if the goal is not to compete against existing MPI distributions, the performance of the solution is comparable with MPI performance, even better in some cases. From the results obtained in the evaluation of this prototype, we conclude that the overhead introduced by the components is not negligible, but inside of the expected. However, we can expect the benefits to grid applications to bypass the generated overhead. Besides, the extended interface may offer users the adequate abstractions to design parallel algorithms in a hierarchical way addressing grid environments.

Keywords: Parallel programming, component-oriented programming, programming model, grid programming, messsage passing, MPI, parallel programming, component-oriented programming, programming model, grid programming.

Passagem de Mensagem Hierárquica Através de um Runtime Baseado em ProActive/GCM

RESUMO

Nos útlimos anos, computação em grade tem emergido como uma forma de utilização de recursos geograficamente distribuídos em múltiplas organizações. Devido ao fato de grids serem altamente distribuídos e compostos por recursos heterogêneos, a computação em grade tem dado importância à requisitos específicos, como escalabilidade, desempenho e a necessidade de um modelo de programação adequado. Vários modelos de programação já foram propostos para a computação em grade. Entretanto, até agora, nenhum deles supriu todos os requisitos. Diferentemente, na área de alto desempenho em clusters, o modelo de passagem de mensagens se tornou um verdadeiro padrão com um grande número de bibliotecas e applicações legadas.

Este trabalho propõe um framework híbrido que combina os altos desempenho e aceitação do padrão MPI, melhorado com extensões intuitivas para permitir aos desenvolvedores o projeto e desenvolvimento de aplicações em grade ou a *gridificação* de aplicações já existentes, com a flexibilidade de um runtime baseado em componentes, modelando uma hierarquia de recursos e suportando a comunicação entre clusters. A solução proposta se baseia na adição de comunicadores MPI e uma API relacionada, a qual oferece um supporte ao desenvolvimento de aplicações que levam em conta a topologia hierárquica de grades computacionais, adequado à desenvolvedores habituados à MPI.

Experimentos realizados com um grupo de applicações com diferentes características (Simulação Baseada no Algoritmo de Monte Carlo, Mergesort e um solver Poisson3D) mostraram que a *gridificação* pode melhorar consideravelmente o desempenho dessas aplicações em ambientes de grade. Ainda que o objetivo deste trabalho não seja competir com distribuições MPI existentes, o desempenho da solução proposta é comparável ao desempenho de MPI, sendo melhor em alguns casos. A partir dos resultados obtidos com o protótipo apresentado, é possível concluir que o custo adicionado pela utilização de componentes não é desprezível, mas dentro do esperado. Entretanto, espera-se que os benefícios para aplicacões de grade devem superar os custos adicionais. Além disso, as extensões à interface MPI oferecem à usuários as abstrações necessárias ao projeto de algorítmos paralelos de forma hierárquica, visando ambientes de grade.

Palavras-chave: parallel programming, component-oriented programming, programming model, grid programming, message passing, MPI.

1 INTRODUCTION

In the past several years, grid computing has emerged as a way to harness computing resources geographically distributed across multiple organizations. Due to its inherently largely distributed and heterogeneous nature, grid computing has enlarged the importance of specific requirements (see section 2.1.3), that are subject of many research projects nowadays.

Initially, research efforts in grid computing focused on providing access to physical resources, including the development of tools for the construction of virtual organizations (VOs), access and allocation of resources. The second step is to offer adequate programming models and execution environments (also referred as frameworks on this document). A new research area therefore emerged, which focused on *programming models and tools* for programming applications which could be efficiently deployed on grids.

1.1 Problematics

Several programming models have been proposed for Grid programming. Nonetheless, so far, none of them met all the requirements, namely dynamicity, scalability and performance. As already mentioned in (FOSTER; KESSELMAN, 1999),

> *Grid environments will require a rethinking of existing programming models and, most likely, new thinking about novel models more suitable for specific characteristics of Grid applications and environments.*

Differently, in the field of high performance cluster computing, the message passing model became a true standard with a large number of libraries and legacy applications. For this reason, the usage of the well known and accepted MPI to develop grid applications have always been investigated in research and industry.

While not a high level programming model by any means, the message passing model lacks in dynamicity and abstractions to program grid applications. Indeed, the MPI standard addresses cluster environments, not having primitives adapted to program multi-site grid environments (PEZZI et al., 2007), that are inherently hierarchical. Contrary to message-passing, a component-based model encompasses most of the programming models proposed to grid programming (MOREL, 2006) (message passing, distributed objects, skeleton-based programming, service-oriented and workflow models) as it provides most of the features presented by other models and, in addition, the capability of encapsulating code. Thus, it should be more adequate to develop grid middlewares.

1.2 Objectives and Contribution

In order to address the problematic of the lack of mechanisms to develop high-performance grid-aware applications, this work proposes a hybrid framework that combines:

- the high performance and high acceptability of the MPI standard boosted with intuitive extensions to enable developers to design grid applications or *"gridify"* existing ones
- with the flexibility of a component-based runtime modeling resources hierarchy and offering support to inter-cluster communication

This approach meets grid programming requirements, offering MPI programmers a straightforward way to execute unmodified MPI applications or develop/adapt their applications/algorithms in a grid-aware hierarchical manner, yet taking profit of legacy high-performance codes.

In practice, the proposed solution relies on the addition of new MPI communicators as an abstraction to deal with the hierarchical structure of multi-site grid environments and a related API, that may offer a support well-suited to programmers used to MPI in order to reflect a hierarchical topology within the deployed application. Moreover, grid related issues are considered in the implementation of the primitives and runtime, in a transparent way for programmers.

The implementation of the prototype is based on standard MPI and the ProActive platform, which offers the adequate support to deploy and execute MPI applications and also offers the reference implementation of the CoreGRID (COREGRID NETWORK OF EXCELLENCE, 2007) Grid Component Model (GCM) (GRID COMPONENT MODEL SPECIFICATION, 2007).

The main contributions of this work include:

1. the definition of intuitive extensions to the MPI standard, addressing hierarchical communication;
2. a support to easily deploy and control the execution of MPI applications in multi-cluster grids;
3. a component-based message passing framework that expresses an overlay structure to support inter-cluster communication;
4. some test applications *gridified* by the used of the proposed extension

The goal of the work is neither compete against existing MPI distributions nor replace the MPI standard, but rather offer an alternative more adapted to the development of grid-aware applications and a runtime supporting the introduced features.

1.3 Document Organization

The reminder of this document is organized as follows:

- Chapter 2 presents some important concepts and important considerations regarding the usage of the MPI standard in grid computing;

- the chapter 3 analyzes some related research projects and tools that are compared later, in the chapter 7, to the work developed;
- the chapter 4 presents the theoretical foundations, principle and specification of this work;
- the chapter 5 shows the main development resources that were used in the development of the prototype;
- the chapter 6 describes the development of the prototype that implements the specification provided in the section 4, taking into account the principles defined on this same chapter.
- the chapter 7 presents the evaluation of this work, which consists in benchmarks, three applications and a qualitative analysis in comparison to related tools
- the chapter 8 presents the conclusion of the work along with some research perspectives created by this work.

2 IMPORTANT CONCEPTS AND POSITIONING

This chapter presents some important concepts related to grid computing and the positioning of this work in relation to the grid definition, characteristics and issues. In order to understand the approach of using message passing in grid environments, we also analyze the main programming models for the development of grid middlewares and applications. After, the MPI standard and its applicability to grid computing is discussed.

2.1 Grid Computing

2.1.1 Definition

Having a complete grid definition is considered important to determine exactly whether a given technology can be considered to be a grid or not (FOSTER, 2002). Since the term *grid* was created, a great number of definitions has been proposed, even by the same authors. One of the widely adopted, proposed by Foster defines a *computational grids* as a system that coordinated distributed resources using standard, open, general purpose protocols and interfaces to deliver nontrivial qualities of service (FOSTER; KESSELMAN, 2003).

The key elements of this definition are the following:

- *Coordinated distributed resources.* A grid integrates and coordinates resources and users that live within different domains, each one regulated by their own usage policies.
- *Using standard, open, general purpose protocols* and interfaces that address fundamental issues as authentication, resource discovery and access. Otherwise, it is an application-specific system.
- *For deliver non-trivial qualities of service*, related for example to response times, throughput, availability and security, and/or co-allocation of multiple types of resources in order to meet complex user demands, so that the utility of the combined system is significantly greater than the sums of its parts.

Some authors consider this definition slightly abstract and use a more specific definition (BAKER; BUYYA; LAFORENZA, 2002). According to this definition, a grid is also defined by the points listed before; however, instead of considering any kind of resource, just *clusters* are focused. This kind of infrastructure is also called by some authors *multi-cluster* or *cluster-of-clusters*. On this work, we refer to grid as a *multi-cluster* infrastructure geographically distributed in multiple sites.

2.1.2 Characteristics

Although there exists several different definitions, a set of characteristics are common (BOTE-LORENZO; DIMITRIADIS; GOMEZ-SANCHEZ, 2004) and important in the context of this work:

- Large scale: a grid middleware must be able to deal with a number of resources ranging from just a few to thousands. This raises the need of scalable solutions to avoid potential performance degradation as the grid size increases;
- Geographical distribution: grid resources may be located at distant places. This characteristic raises the problem of dealing with different network characteristics (latency, bandwidth) and the resultant impact in performance;
- Heterogeneity: a grid hosts both software and hardware resources that can be heterogeneous: data, files, software, hardware configuration and networks;
- Resource sharing: grid resources usually belong to many different organizations that allow users to access them. For that reason, the collection of resources can be seen as a great shared resource. This assumption lead to necessity of applications to adapt to available resources;
- Multiple administrative domains: each organization may establish different security and administrative policies under which their owned resources can be accessed and used. As a result, the already challenging network security problem is complicated even more with the need to cope with different policies.

2.1.3 Grid Programming Issues

Due to the inherent characteristics discussed in the previous section, grid environments increases greatly the emphasis on some issues (BERMAN; FOX; HEY, 2003), namely: portability, interoperability, security, fault tolerance, the need of an adequate programming model and performance.

2.1.3.1 Portability, Interoperability and Adaptivity

Current high-level languages allow code to be processor-independent. Considering that grids are potentially heterogeneous, grid programming tools should also enable the applications to have the same portability (BERMAN; FOX; HEY, 2003). This can means architecture independence in the sense of languages interpreted by virtual machines (e.g. Java), but it also can mean the ability to use different services at different locations with equivalent functionality. Such portability is necessary to cope with heterogeneous resource configurations.

Another important point is the idea of interoperability, that stands in the notion of the capability of two or more components or implementations to interact. In grid environments, interoperability have a close relation with the notion of an *open and extensible grid architecture* implies a distributed environment that may support protocols, services and application programming interfaces (APIs) (FOSTER; KESSELMAN, 2003).

Related to portability and interoperability, another important issue is adaptivity, as the capability of a grid application to adapt itself to different configurations, depending on the resources. This could occur at start time or at run time due to

changes on the application requirements, resources availability or fault recovery, for example.

2.1.3.2 Security

Grid applications usually run across multiple administrative domains. When resources are shared across organization boundaries, security is an important issue because it permits a domain to be accessed by other domains, and this may be exploited by malicious users. Also, the security in application level plays an important role.

Security requirements within grid environments are driven by the need to support scalable, dynamic, distributed virtual organizations (VOs) (FOSTER; KESSELMAN, 2003), that potentially can be composed by several domains. The VO works as a policy overlay which coordinates the outsourced policy in a consistent manner to allow the resource sharing and use.

2.1.3.3 Fault Tolerance

Naturally, as the numbers of resources involved increases, so does the probability that some host or link will fail during the computation. For that reason, grid applications may have the possibility to check run-time faults in communication and/or computing resources and provide actions to recover or react to these fails.

At the same time, some grid environments are composed by shared resources, that may arrive and leave at any instant. For this reason, fault tolerance should be an integral part of grid programming.

2.1.3.4 Performance

One of the the many usages for grid environments has been high performance computing (e.g. scientific applications and simulations) and data storage. In these applications, performance is usually one of the strongest needs.

A second issue related to performance is *scalability*, as the degree to which a system or application can handle increasing or decreasing amount of resources, without significant performance degradation. To keep scalability in heterogeneous and dynamic environments is a real challenge. Indeed, the need of a reliable performance for some application may prevent the use of grid environments for depending on the purpose.

2.1.3.5 Programming Models

Besides of an infrastructure that provides access to resources, authentication and security, grid programming tools also must support programming models and abstractions to simplify the production of applications.

An adequate programming model must be flexible, so that the programmers could easily express their algorithms, easy to use and understand and also cope with the grid characteristic. If possible, this model must also be compliant with existing technologies, in a sense that legacy software could be useful to develop grid applications.

To find a program model that fits in all these requirements has been proved to be another great challenge (FOSTER; KESSELMAN, 2003) and, for this reason, many programming models have been proposed and adapted last years. The follow-

ing section presents some of the programming models that have been proposed by research and industry.

2.1.4 Candidates Programming Models for Grid Computing

As previously referred, the large number of requirements and characteristics, makes the development of grid applications a difficult task. So, many programming models have been proposed to ease this process. The following subsections present an overview of the main programming models that are being used to develop to grid solutions and applications. Some of the concepts presents on these subsections are important on the definition of the approach to develop the proposed work.

2.1.4.1 Message Passing

The message-passing paradigm is one of the most popular programming models, especially in high-performance computing and for scientific applications. The main goal is provide user with low-level primitives and abstractions for point-to-point, collectives communication and synchronization.

This paradigm is very close to operating system mechanisms. So, it does not provide the high-level abstractions of the other models presented on the next sections. For this reason, it is said that it has a more complex usage (FOX, 2002). However, the model offers a more accurate control of the communication process, that is considered one of the main critical performance issues.

On the other side, the lack of high-level abstractions to address issues like hierarchical topology, dynamicity and heterogeneity inhibits the direct usage of message passing in grid environments. However, the high acceptance of this model by the developer communities and the large amount of legacy applications have created a great interest on the application of this model to grid computing. Some research efforts (PEZZI et al., 2007; CERA et al., 2007) have indeed tackled issues like hierarchy and dynamicity.

Most of projects that aims at the usage of explicit message-passing in grids adopted the MPI as the programming interface. This choice relies in the spread usage of this standard, clear interface and high performance of its implementations. The section 2.2.2 and 2.2.3 present, respectively, the main benefits and drawbacks of the usage of message passing in grids.

Some projects that take this approach are PACX-MPI (GABRIEL et al., 1998), MPICH-Madeleine (AUMAGE; MERCIER; NAMYST, 2001; AUMAGE; MERCIER, 2003), MPICH-G2 (KARONIS; TOONEN; FOSTER, 2003) and H_2O MPI. More details about these projects can be found in the section 3.

2.1.4.2 Remote Procedure Call (RPC)

The concept of Remote Procedure Call (RPC) has been widely used in distributed computing for many years. It also support process interaction in a distributed environment by extending the notion of a procedure call to operate across the network. Also called *one-sided message-passing*, the RPC model offers a higher abstraction level that not requires an explicit receive operations.

In addition to distribution, RPC implementations also address heterogeneity by using neutral interface description languages. However, these models assume the knowledge about the name/identifier, address, and the existence of the end-

points. Also, the syntax and semantics of the interface are known at compile-time. Considering the dynamic nature of grid environments, the assumption of a complete previous knowledge of address may not be present as well as the existence of endpoints. For that reason, the need of extra mechanisms to address such issues arises.

Some examples of RPC middlewares for grid computing are GridRPC and OmniRPC. GridRPC is an RPC model and API for grids that uses Globus Toolkit to offer dynamic resource discovery and scheduling, security and fault tolerance. OmniRPC is a thread-safe RPC facility on top of Ninf to broke remote procedures, associating them with remote stub interface information at run-time.

Worthy to notice that the related tools do not use exclusively a RPC approach to address grid programming requirements, being necessary the usage of others tools (Globus, Ninf) to deal with such issues. Besides, RPC is more adapted to client-server interactions, while more complex interactions are usually required to develop tightly coupled applications.

2.1.4.3 Distributed Object Model

Thanks to its high-level and programming concepts, the object-oriented paradigm is currently a widely used programming approach. Distributed communications between objects are easily done through remote method invocations, either through a standardized extension of the language (such as Java RMI) or through a tier middleware layer (CORBA). Also, the portability of some object-oriented languages , such as Java, ease the development of distributed objects middlewares

Some examples of middlewares that adopt distributed objects as the model to develop grid applications are GridGain, Satin and ProActive. Satin extends the Java language for providing parallel execution of method invocations, it targets divide-and-conquer programs by offering dedicated constructs (spawn and sync) and automatically load-balances the invocations. Satin uses an optimized communication layer called Ibis. ProActive (BADUEL et al., 2006), is a grid middleware for parallel, distributed, and concurrent computing based in the idea of *active objects* , that also features mobility and security in a uniform framework. More about the ProActive middleware is presented in the section 5.1.

2.1.4.4 Skeleton Model

Skeletons are high-level and parametrized algorithmic patterns, introduced by Cole *et. al* (COLE, 1989). Complex applications can be designed with highly structured interactions due to the composition of basic skeletons such as farm, pipe, map, etc. Skeleton facilitates a top-down design approach, where a partially-functional system with complete high-level structures is designed and coded.

Several frameworks offer skeleton programming facilities for grid computing. ASSIST (acronym for Software development System based upon Integrated Skeleton Technology) (ALDINUCCI et al., 2004) provides a high-level language with a compiler, as well as a runtime support. ASSIST also provides interoperability with CORBA and plans interoperability with Globus for accessing grid services. HOCSA (DUNNWEBER; GORLATCH, 2004) is another skeleton framework that offers interoperability with grid services through Globus by using higher-order components (HOC) to emphasize the possibility of composing skeletons.

Current work around skeletons for Grid computing focus on structured and optimized distributed and parallel programming in order to achieve high performance.

Grid computing offers a wider diversity of programming challenges and the latest developments in grid skeletons programming seem to converge with another programming model: component-based programming (section 2.1.4.6) (MOREL, 2006).

2.1.4.5 Service-Oriented and Workflow Models

Service based frameworks intend to provide interoperability, on-demand access and loose-coupling, as a way to achieve scalability. Also, services are seen as a way to simplify design, enable code reuse, and facilitate integration of tier products and collaboration among companies.

Services, by themselves can be defined as means to access the grid infrastructure (FOSTER; KESSELMAN, 1999). Nonetheless, the orchestration of services into workflows requires the usage of an adequate workflow language, and a workflow engine to coordinate the participating entities at runtime.

Many workflow languages are available for grid computing (YU; BUYYA, 2005), but a *de facto* standard does not exist yet. However, some industrially established workflow standards such as BPEL are extensible enough to suit the needs of grid computing.

Workflow composing (either automatically from a program or by graphical composition) is said to be simpler and better suited than lower-level coding and assembly for scientists or other grid applications designers who are not expert programmers. However, grid services are not suitable for tightly synchronized applications, because of the communication overhead of XML/SOAP mechanisms.

In the last few years, a big effort has been out on the standardization of services for grids, mainly in the context of the Open Grid Services Architecture (OGSA).

2.1.4.6 Component-Based Models

Component-based programming is another programming model used for grid computing. One of the most accepted definitions describes a software component as an unit of composition with contractually specified interfaces and explicit context dependencies only. Besides, a software component can be deployed independently and be subject to composition by third parties (SZYPERSKI; PFIZER, 1996).

The idea behind using a component-based approach is that this model addresses increasing software complexity and changing requirements by enabling the construction of systems as an assembly of reusable components. Because of the modularity and extensibility, the usage of such approach may fit well actual grid systems and their many issues (2.1.3). Current component models for grid Computing include the Corba Components Model (CCM), Common Component Architecture (CCA) and Grid Component Model (GCM) .

CCM is defined by the Object Management Group and extends the CORBA distributed object model, providing a similar support to distribution, heterogeneity and security. It also supports dynamic instantiation and runtime customization of components. However, CCM inherits some of the limitations of CORBA, like the requirement of a previous knowledge about interfaces and interactions. CCA, that is currently coordinated by the CCA Forum defines a component model especially for scientific applications. The model primarily addresses the heterogeneity and the separation of concerns. The CCA component model does not address failure or security and assumes all components are trusted. GCM, defined by the CoreGrid project defines a lightweight component model for the design, implementation and

execution of grid applications. The reference implementation of the model is named ProActive/GCM and addresses programmability, interoperability, code reuse and efficiency by means of a component framework. More information about ProActive/GCM is presented in section 5.4.

2.2 MPI and Grid Computing

The relation between the MPI standard and grids as an infrastructure to execute high performance applications has never been straightforward. Created to ease the development of applications to parallel machines (such as MPPs and clusters), the MPI standard has always focused in performance and not so much in characteristics like portability, fault tolerance, etc. However, at some extent, MPI includes some characteristics and primitives that might fit some of the requirements of grid applications, but not all. This section presents the two versions of the MPI standard (1.2 and 2.0) and a discussion about the main benefits of using the MPI standard in grid environments in contrast with the main constraints that prevent an unrestricted usage of MPI in actual grid infrastructures.

2.2.1 The MPI Standardization

The MPI standardization effort (MPI Forum, 1994), initiated in 1992, involved about 60 people from 40 organizations, mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry.

Before MPI, PVM (SUNDERAM, 1990) was the reference on message passing environment, but with a stronger focus on resources/process management, dynamicity, the idea of a virtual parallel machine and transparency. On the other side, MPI focus in performance, a clear interface featuring a more powerful support to collective communication and different parallel machine architectures, from shared-memory multiprocessor machines to clusters. Also, the support to fast interconnection networks has being one of the main keystones.

2.2.1.1 MPI 1.0 and 1.2

The version 1.2 of the MPI standard, launched in 1997 present just some small modifications to the 1.0 and 1.1 version of the standard. In fact, just the errata of the previous version and some clarifications about intercommunicators were included.

The main features of the MPI standard includes (MPI Forum, 1994):

- Point-to-point communication: are the base of the communication processes, having different modes (synchronous/blocking, buffered/unbuffered) so that the user could explicitly specify the communication process in order to obtain a better performance on the communication process;

- Collective operations: the strong support to collective communication is in the core of the MPI functionalities. Many modes of collective communication and barriers are supported;

- Process groups: the MPI processes are grouped through the communicator abstraction. Besides of pre-defined communicators, the user has the possibility

of creating new communicators and perform operations (merge, split, ...) on them;

- Communication contexts: besides of groups of process, the standard defines communication contexts, that are an abstraction associated with communicators, where optimizations like support to high performance networks can be useful;

- Process topologies: the standard defines a set of primitives that enable the creation of communicators and assignment of process ranks according to specific topologies. Some of these topologies are pre-defined like cartesian, torus, etc. But it is also possible to define new topologies by means of processes graph;

In order to limit the scope of the standard, there is a explicit mention that the standard does not include:

- shared-memory operations;

- operating-system related functionalities, for example, interrupt-driven receives, remote execution, or active messages;

- debugging facilities;

- explicit support to threads;

- support for task management

- I/O functions.

2.2.1.2 MPI 2.0

Since the creation of the MPI standard, some important features are known to be missing: use of dynamic resources, co-existence with thread programming (thread-safety) and memory management (MPI Forum, 1997). Besides, the evolution of parallel systems from static and homogeneous systems (i.e. clusters) to dynamic, heterogeneous and multi-domain systems (i.e. grids and it many definitions) have shifted the need on some of these functionalities.

For this reason, together with the creation of the version 1.2 of the standard (1997), the version 2 was proposed. The main advances included by the version 2.0 to the MPI interface includes:

- one-sided communication: the new version of the standard includes support to Remote Memory Access (RMA) via one-sided communications (put and get operations), somehow relaxing the idea of need of a matching receive to a send operation;

- dynamic process creation and management: MPI applications are now capable of creating and managing new MPI processes, but the abstraction of static communicators is still valid and once a communicator is built it behaves as specified in MPI-1. Indeed, the new standard creates the possibility of creating new bindings between processes other than those created at the beginning of the application or creation of new process. This is done through publish/connect primitives;

- parallel I/O: useful functionalities were included offering primitives to deal with transparent parallel access to data and files, leveraging the issues related to file sharing;
- thread support: the standard now defines some minimal requirements for thread-compliance so that MPI process could interoperate more easily with thread libraries in a safe and controlled way;
- extended collective communication operations: besides of the existing operations, the MPI-2 includes new operations over communicators like cloning and linking communicators. Besides, the idea of intercommunicators became useful to support the communication with process created dynamically.

Even if ten years have passed, just a few stable MPI distributions include a complete support to this version, such as LAM-MPI, NEC, MPICH2 and OpenMPI. Some others, like FT-MPI and SGI implements just some parts of the specification.

The following sections discusses in more details some of these features and their applicability on grid environments.

2.2.2 Benefits of the MPI Standard for Grid Computing

Many aspects have motivated the usage of the MPI standard to develop grid applications. The idea is that the concepts and characteristics that made MPI a standard for cluster computing might be used or adapted to grid computing. Some of these aspects are presented in the next subsections.

2.2.2.1 MPI is a de-facto standard

Since MPI was launched, it superseded the current standards on high performance computing (notably PVM), becoming one of the strongest standards to develop parallel applications. At a moment of software crisis in parallel computing, with many interfaces being offered by different vendors, its clear yet powerful interface have motivated many academic projects (like the MPICH, LAM, etc.) and the industry (Intel, SCI, Cisco, etc) to develop and maintain implementations of the standard as well as numerical libraries and tools (WILKINSON; ALLEN, 1999).

For this reason, a large number of legacy MPI applications exist nowadays, an many of them consumed years of work to be done. Indeed, the community of users is very large and stills growing up. Thus, the idea of taking profit of all the knowledge and work done encouraged the usage of the standard in grid computing.

2.2.2.2 Support to heterogeneous environments

As previously referred in the section 2.1.2, one of the main characteristics of grid environments is the heterogeneity of resources (computers architecture, network, softwares, etc.). In this sense, portable solutions are highly desirable for the use of a grid as an unified resource.

The MPI standard defines minimal portability requirements. However, in native implementations of the interface (in opposition of Java implementations, such as MPIJava, JMPI, etc.), such property is not acquired automatically, and depends, at least, on the compilation of specific versions for each of architectures involved. By doing so, MPI processes running on heterogeneous resources should be capable of interoperate as they were in an homogeneous system.

At the core of the MPI implementation, some abstractions like the MPI datatypes and packaging primitives are offered to ensure portability. Besides, MPI stands on the top of a library that implements this interface accordingly to the specification. However, the interoperability between different vendor implementations of the standard is not the rule.

2.2.2.3 Dynamic process creation and management features

The MPI-2 standard (MPI Forum, 1997) brought many new features to reduce the gap between the MPI standard and dynamic, heterogeneous environments such as grids. However, there is not an clearly intention to support grid programming in the standard.

One of the main improvements introduced in the version 2.0 of the interface is the support to dynamic process management. Due to this newly introduced feature, it became possible to include dynamically resources and manage (create, change) bindings between MPI processes.

Despite of the usefulness of these features, this solution in not enough do deal with dynamic environments. We discuss better this point in the section 2.2.3.

2.2.2.4 High performance and stable implementations

Many stable implementations of the MPI standard exist today. While some of them offer a complete implementation of the standard (version 1.2 or 2.0), others focus on specific features. Some known examples are the FT-MPI that focus on fault-tolerance and ROMIO, that focus on I/O performance. Some other are vendor specific and intend to exploit hardware capabilities at most.

In this sense, an important remark is that multi-site grid infrastructures are a coupling of a few (or many) clusters geographically distributed and that, usually, such resources are internally connected with special network connections (e.g. Myrinet, SCI, FiberChannel, etc.) for the sake of better performance. Many MPI implementations offer support to these networks, so we can expect an improvement of the performance of the communication process.

2.2.3 Limitations of the MPI Standard for Grid Programming

If on one side many characteristics and features of MPI make message passing an interesting programming model to develop grid applications, on the other side, there are a number of other characteristics and limitations that prevent the direct usage of MPI over computational grids. These limitations are addressed by some projects (Chapter 3) and analyzed in more details in the following subsections.

2.2.3.1 MPI is too static in design

One of the main characteristic of grid environments is the dynamic offer of resources. In order to cope with this characteristic, applications must adapt themselves to changes in the environment, even if changes might be defined in reservation time. Thus, the tools used to develop and execute grid applications should be capable of expressing this dynamic behavior.

This is not the case of the MPI standard. First, because the entire communication process is based on the idea of communicators that are static structures which cannot be changed after they were created. Actually, MPI includes just a few

primitives to create, merge and split existing communicators, but it does not offer an adequate support to support changes in the environment.

Another strong aspect that shows the static design of MPI is the semantic of the message passing, where both sides of the communication process must explicitly call primitives for sending and receiving messages. Indeed, these primitives must, in general, identify the source and destination through identifiers.

The version 2.0 of the standard includes primitives to perform dynamic creation (MPI_Comm_Spawn) and control of processes and one-sided communication (MPI_Put and MPI_Get, etc.). However, the usage of resources included on-the-fly depend on the creation of a communicator, that is itself a static entity.

2.2.3.2 MPI is a low-level interface

Different from most of the tools created recently for distributed computing, MPI provides a low level interface. This means that users must take care of many issues others than the application itself: memory management, buffers, communication modes, management of communicators, etc.

If, on one side, a low-level interface may improve performance and expressiveness, on the other side, it is more error prone and usually difficult to debug. When running on a real grid infrastructure, composed by a large number of heterogeneous and dynamic resources these problems tend to increase.

Also, modularity, encapsulation and code reuse are not emphasized, as SPMD is the main approach to build MPI applications

2.2.3.3 MPI does not cope with some grid issues

Because of the geographical distribution and differences on the network characteristics (latency/bandwidth), from LANs to WANs, a straightforward way to model a multi-site grid is through a hierarchical composition. MPI, on the other way, suppose flat environments. Besides, MPI presupposes direct all-to-all node access, while grid infrastructures with limited connectivity are commonplace.

Grid environments have a lot of other specific needs, like resources access/allocation and fault tolerance. As the standard does not have anything related to these aspects, they must be addressed by third parties tools.

In fact, there is no complete solution to develop MPI grid-aware applications, and solution to many of these requirements already exists in grid middlewares. However, an additional integration is needed to make MPI run according to those mechanisms: deploy MPI processes in multiple domains, support failure in nodes and communication between nodes that don't have access to each-other.

2.3 Chapter Conclusion

This chapter presented some important concepts related to grid computing and different programming models. Due to the many existing grid definitions, the concept that describes grids as geographically distributed clusters is considered in the remainder of this document. On this context, the characteristics of being a large scale environment, geographically distributed, inherently heterogeneous and composed by shared resources are focused on this work.

The main grid programming requirements presented on the section 2.1.3, addresses by this work are:

- Portability, Interoperability and Adaptability: in general, it is difficult to know in advance the resources available for the execution of a given application. Besides, it is important to an application to be able to run efficiently on a different set of resources, scaling as much as possible. By providing an specific API, we expect to make programmers capable to take into account available resources on the deployed applications. On this work, portability is not considered a constraint because we suppose resources are homogeneous in software. However, the solution provides interoperability among multiple MPI distributions as a manner to take profit of different hardware and network profiles.

- Performance: performance is always a constraint in distributed systems and this problem is tackled from different perspectives:

  - offer an API adapted to the development of grid-aware applications;
  - usage of native MPI on inner-cluster communication, to avoid crosscutting software layers and take profit of MPI optimizations;
  - optimizations on the inter-cluster communication process.

- Programming Model: In the development of this work we consider different programming models for the application level and framework. For the interface with the environment (API) we decided to follow the MPI approach for all the reasons explained in the section 2.2.2, but an advanced support to hierarchical communication tend to solve some of the current limitations of MPI for grid computing. Differently, for the runtime, we opted by a component-based infrastructure, for the following reasons:

  - components offer a clear separation of concerns, which makes easier to experiment different solutions and optimizations
  - components can provide an easy encapsulation of MPI codes
  - the ProActive/GCM implementation has a reasonable performance and may come up with some important features, such as interoperability with grid tools, deployment, file transfer, tunneling of messages through ssh and built-in collective interfaces.

A deeper explanation of these principles and choices will be presented in the chapters 4 and 6.

3 RELATED RESEARCH PROJECTS AND TOOLS

Many of the reasons that motivate this work, also motivated several research projects: the spread usage of the MPI standard, clear interface and high performance of its implementations. Even with completely different approaches, the related works intend to address grid issues in order to make MPI an interface to develop grid applications.

The following sections present some of these projects and their approach to enable MPI over computational grids.

3.1 PACX-MPI

The PACX-MPI (KELLER et al., 2003), developed in the High Performance Computing Center of Stuttgart (HLRS), is an implementation of the MPI standard, based in MPICH, which aims at supporting the coupling of high performance systems (*clusters*) distributed in a grid environment.

The PACX-MPI project defines three major design goals (BEISEL; GABRIEL; RESCH, 1997):

- Provide the user with a single virtual machine. No changes to the code are necessary at all;
- Use highly tuned MPI for internal communication on each MPP;
- If possible, to use fast communication for external communication.

The main idea behind these goals stands in the fact that vendor MPI implementations should provides optimal performance and, for that reason, they must be used as much as possible. Just in the case of impossibility of using it or for handling external communication (e.g. between MPPs), the PACX-MPI communication layer is used.

So far, MPI-1.2 standard is fully supported as well as some parts of the MPI-2. However, useful features of the MPI-2 standard, like dynamic creation and management of process, parallel I/O and the extensions to the group communication (intercommunicators, management of groups) still missing.

The project does not define any extensions to the MPI standard. Thus, legacy applications can be executed in a grid as an unified computational resource. On the other side, one can expect a decrease in the communication performance, as there is no way of defining beforehand the physical location of the process. The access to resources, its allocation and management are also not on the scope of the project.

The following subsections will present more information about the PACX-MPI architecture as well as its main features.

3.1.1 Architecture

PACX-MPI is implemented as a library that stands between the application and a local MPI implementation. When the application call MPI functions, they are intercepted into PACX-MPI, that verifies the need for contacting the outside world. If yes, the communication is made through TCP *sockets*. On the other side, the library passes the calls unchanged to the local system.

For doing so, the PACX-MPI includes *daemons* on each MPP. These *daemons* are responsible for forwarding messages from inside to outside of MPPs and vice-versa. The nodes responsible for receiving the *daemons* are previously defined and must be configured in order to meet cross-firewall configurations. The PACX-MPI infrastructure is organized as presented in fig. 3.1. We can see one *daemon* by MPP and the PACX-MPI library in each of the process so that messages could be forwarded to PACX-MPI daemons.

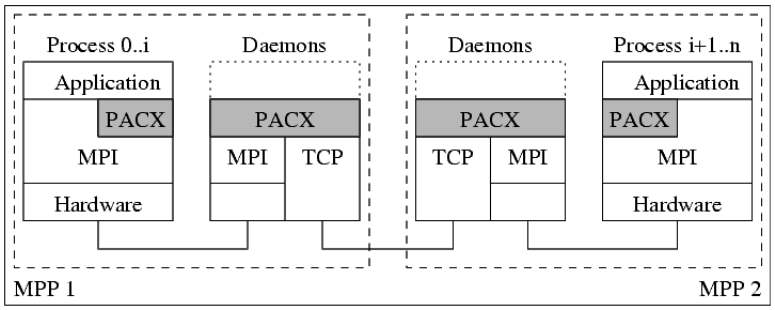

Figure 3.1: PACX-MPI Architecture

3.1.2 The Main Features

3.1.2.0.1 Private Network and Tunneling Support

By using *daemons*, PACX-MPI offers the possibility of coupling resources enclosed on private networks or behind firewalls.

Through the communication forwarded by the *daemons*, PACX provides to the users the idea of a global MPI communicator composed by all process. As a consequence, these process have, at the same time, a local rank and a global one.

The PACX library decides automatically when to use the local ranks or global ranks depending on the topology and resources used on the application. The figure A.1 shows how the global and local ranks are organized.

As previously referred, this feature depends on the user configuration of tunnels between gateway hosts.

3.1.2.0.2 Optimization of Global Communication

For the obtention of a better performance on collective operations, two algorithms are used within PACX-MPI (KELLER et al., 2003):

- Linear algorithm: In this algorithm, the root receives from each node its parts of the message in a linear order.

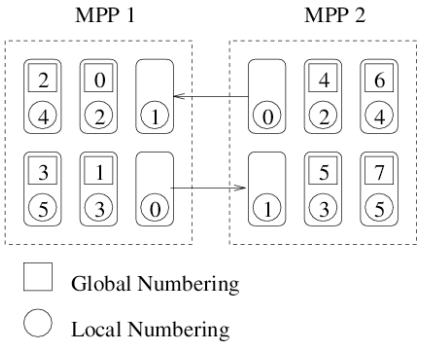

Figure 3.2: PACX-MPI Rank Scheme

- Host-based algorithm: This algorithm is split into two parts: a local part where the root node collects data from all other nodes on its machine and a global part where the global root node collects data from local root nodes.

For short messages, the host-based algorithm is faster than the linear one, since it minimizes the number of messages over the slower link between the machines. On the other side, with the increasing of the message size, the linear algorithm will be faster than the host-based algorithm, since it avoids the additional internal communication steps (which are not negligible)

3.2 MPICH-G2

The MPICH-G2 (KARONIS; TOONEN; FOSTER, 2003), developed in the Computer Science Department of the Northern Illinois University, together with the Argonne National Laboratory, is a complete grid-enabled version of MPICH that uses services provided by the Globus Toolkit to enable MPI for *grid* environments.

The idea behind the MPICH-G2 is to hide heterogeneity using Globus services for purposes like authentication, authorization, process creation, process monitoring and control, communication, redirection of IO and remote file access. As a result of this integration, users can run MPI programs across multiple sites, using the same primitives and even the same commands that could be used on a parallel computer, like a *cluster*.

The following subsections present the MPICH-G2 building blocks, its architecture and also the main features offered by this tool.

3.2.1 Building Blocks

The MPICH-G2 implementation has two main building blocks: the MPICH implementation of the MPI-1.2 standard and the Globus Toolkit for grid computing.

3.2.1.0.3 MPICH

MPICH (GROPP et al., 1996) is one of the most popular implementation of the MPI standard, developed as a collaborative effort between Argonne National Laboratory and Mississippi State University. It implements the MPI-1.2 standard and parallel I/O functionality defined in the MPI-2 standard having a special focus on a high performance and portability. Since the version 2, the MPI-2.0 standard is supported. Nonetheless, the MPICH-G2 is based in an older version.

MPICH derives its portability from its interfaces and layered architecture. The top layer consists of the standard MPI interface. Beneath this interface, there is the MPICH layer that implements the MPI interface independent of the network devices or process management system. The lower layer, that treats the network communication and process control, is defined through an *Abstract Device Interface* (ADI). Actually, an implementation for a particular platform is, in fact, an implementation of the ADI, that is much simpler than MPI.

3.2.1.0.4 Globus

The Globus Toolkit is a collection of software components designed to support the development of applications for high-performance distributed environments, or *grids* (FOSTER; KESSELMAN, 2003). The toolkit comprises a set of protocols for interacting with remote resources, APIs to invoke these protocols, higher level libraries, services and tools for management of the grid environment.

The toolkit addresses issues of security, information discovery, resource and data management, communication, fault detection, and portability. Through the use of such features, MPICH-G2 enables the transparent execution of MPI application in *grid* environments.

The current Globus version (GT4) is mostly based in web services for the sake of having a better interoperability. Differently, MPICH-G2 is based in the GT2 that is mainly composed by native modules and have a better performance.

For the MPICH-G2 implementation, the following components of the Globus Toolkit have major importance:

- Resources Specification Language (RSL): Specification language for describing resources and specifying requirements of applications.
- Grid Security Interface (GSI): Interface that is responsible for the management of credentials that are used to authenticate the user on each site.
- Dynamically-Updated Request Online Coalocator (DUROC): responsible for scheduling processes across specified computers. This service interacts with several job schedulers (PBS, OAR, LSF, etc.)
- Grid Resource Allocation Management (GRAM): a set of APIs and protocols that makes possible to start and subsequently manage a set of subcomputations, one for each computer, previously allocated by the DUROC service.
- GlobusIO: a set of APIs for tunneling communication with mechanisms of tunneling and data conversion (Globus DC), useful in firewalled and heterogeneous grid environments.

3.2.2 Architecture

MPICH-G2 has a layered architecture composed by *Globus* services, MPICH and a implementation of a virtual device called *globus2*, as shown in fig. A.2.

The lower layer is composed by Globus services that are responsible for resource allocation (GRAM), authentication (GSI) and communication when native communication is not possible (GlobusIO). On the top of this service, there is the MPICH

implementation and its *Abstract Device Interface* (ADI) that is implemented using of various *Globus* APIs. As explained above, on the top of the MPICH layer there is the MPI standard and also extensions included by the MPICH-G2.

MPICH-G2, in fact, consists of an implementation of the ADI by means of a virtual device known as *globus2* and also some MPI attributes that are used for acquiring topology information.

The architecture is shown in fig. A.2.

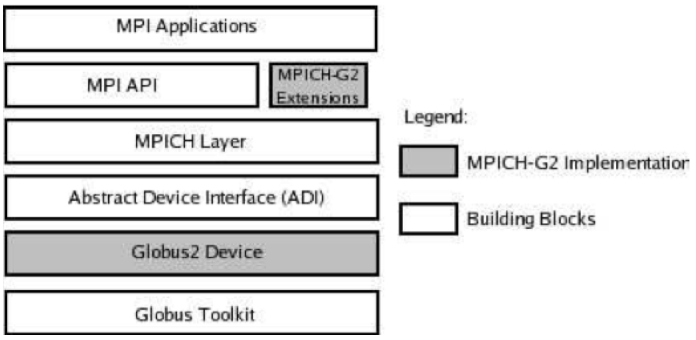

Figure 3.3: MPICH-G2 Architecture

3.2.3 The Main Features

3.2.3.0.5 Startup and Management of Heterogeneous Environments

MPICH-G2 uses a range of Globus Toolkit services to address various issues that arise in heterogeneous, multi-site grid environments, such as cross-site authentication, the need to deal with multiple schedulers with different characteristics, coordinated process creation, heterogeneous communication structures and I/O.

Prior to startup, it is necessary to create a RSL script that defines the resources to be used and the GSI interface is responsible for the authentication through the credentials previously issued by grid authorities. Once authenticated, the user can use an `mpirun` command to request the creation of the MPI processes.

From the side of MPICH-G2, the DUROC library is used to contact the GRAM server that initiate MPI processes on the assigned hosts. If necessary, GRAM also uses GASS services to stage execution and redirect I/O, in order to hide aspects related to location.

Once the application has started, MPICH-G2 also selects the most efficient communication method available, using either native MPI communication or globus communication (GlobusIO), that includes mechanisms for data conversion (Globus DC).

The figure 3.4 shows a scheme of this mechanism.

3.2.4 High Performance Heterogeneous Communication

Different from its previous implementation (MPICH-G (FOSTER; KARONIS, 1998)), that have all the communication made by the *nexus* library (FOSTER; KESSELMAN; TUECKE, 1996), the MPICH-G2 handles itself all the communication directly. According to (KARONIS; TOONEN; FOSTER, 2003), this has increased greatly the bandwidth usage because of the reduction of intermediary copies. Also, a reduction on the latency became possible due to an adaptive pooling

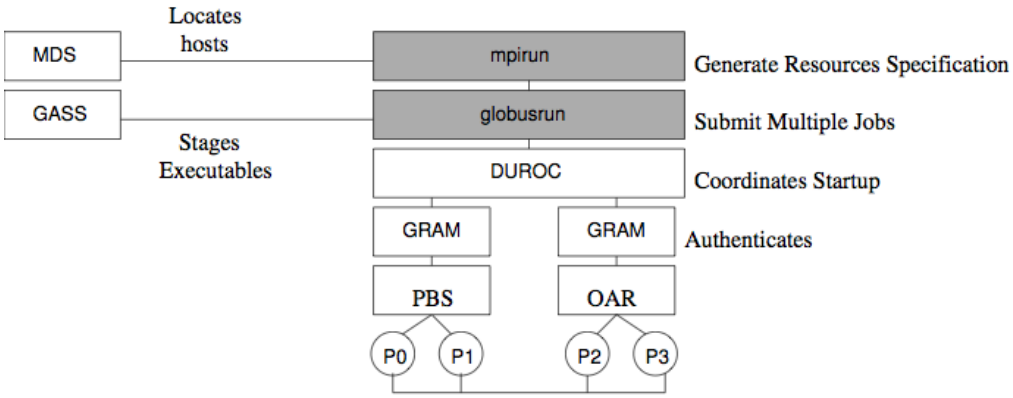

Figure 3.4: Startup of MPICH-G2 showing the various Globus components involved to hide and manage heterogeneity, "OAR" and "PBS" are different local schedulers that may be involved

protocol, i.e. instead of trying to use different protocols (TCP, vendor MPI, ...), the new implementation uses information provided by the source/destination of the messages in order to know which means of communication to use.

Besides of the point-to-point heterogeneous communication, MPICH-G2 offers topology aware collective operations through the use of topology discovery mechanism, that will be explained better in the next subsection.

3.2.4.0.6 Application-level Management of Heterogeneity

Although the heterogeneity aspects are hidden to the programmer, it is important to know about it in order to get a better performance. Once the MPI application has started, the process are just distinguished by their ranks. Although it is desirable from a programming viewpoint, this makes difficult to write programs that exploit aspects of the underlying physical topology, like avoiding expensive inter-cluster communications. For that reason MPICH-G2 associates to the attributes of the MPI communicators information about topology.

These attributes can be used to adapt the application to the underlying topology at execution time. However this is not a trivial work.

3.3 MPICH/Madeleine

MPICH/Madelaine (AUMAGE; MERCIER, 2003), developed in the Bordeaux unity of the Institut National de Recherche en Informatique et en Automatique (INRIA) have the same approach adopted by MPICH-G2 (section 3.2), developing a device called *ch_mad* that implements the MPICH *Abstract Device Interface* (ADI).

Instead of using a grid *middleware* for leveraging grid issues, MPICH/Madeleine intends to provide an implementation that supports high performance communication protocols (Myrinet, Giganet, SCI, etc.) by the use of a library called Madeleine.

Actually, just the MPI-1.2 standard is supported. Support to the MPI-2 standard depends on newer MPICH version that provide such features.

Access and allocation of resources, creation and management of process are put aside and must be treated by other tools. Some scripts might ease this process, but

just for specific cases.

The following subsections present the main MPICH/Madeleine building blocks, its architecture and also the main features included in this tool.

3.3.1 Building Blocks

Two main building blocks compose MPICH/Madelaine: the MPICH implementation of the MPI-1.2 standard and the Madeleine communication library.

3.3.1.0.7 MPICH

As presented in the section 3.2.1.0.3, a grid-aware implementation of the ADI is capable of enabling unmodified MPI application to run on top of grids.

MPICH/Madeleine approach intends not only to allow to run applications on top of heterogeneous architectures (such as clusters of clusters) but also allow application to access all the communication facilities available between each pair of hosts. For that reason, when the ADI is requested to send a message, the appropriate device is selected and after that, the most suited exchange protocol is chosen.

Different from current ADI implementations that are not capable of using two different networks (e.g. Myrinet and SCI) together, the MPICH/Madeleine project did not subscribe to the MPICH philosophy of building a multi-device implementation. Instead, what is provided is a single-device implementation of MPICH on top of a multi-protocol interface called Madeleine, that is presented in the next subsection.

More information about the MPICH structure and implementation can be seen in the section 3.2.1.0.3.

3.3.1.0.8 Madeleine

The Madeleine programming interface (AUMAGE; MERCIER; NAMYST, 2001) provides a small set of primitives to build RPC-like communication schemes. Basically, this interface provides primitives to send and receive *messages*, and several *packing/unpacking* primitives that allow the user specify how data should be inserted and extracted from messages. These primitives intend to provide communication, within heterogeneous environments, in a transparent way.

Madeleine also aims at enabling an efficient and exhaustive use of underlying communication software and hardware functionalities. It is able to deal with several network protocols within the same session and to manage multiple network adapters for each of these protocols. The library can dynamically switch from one protocol to another, according to its communication needs in order to meet the network high performance capabilities.

3.3.2 Architecture

The MPICH/Madeleine architecture is mainly based on the MPICH layered implementation, as described in the section 3.3.1.0.7. In the MPICH structure, just the lower communication layer (ADI) is re-implemented by MPICH/Madelaine.

Three different MPICH devices are concurrently used to handle communication, each one dedicated to a different type of communication that takes place within a *cluster of cluster* (grid) composition:

- *ch_self* device: responsible for handling *intra-process* communication

- *smp_plug* device: responsible for handling *intra-node* communication (for SMP nodes)
- *ch_mad* device: responsible for handling any *inter-node* communication

The Madeleine library is capable of choosing the best device to use and exchange as needed.

The *ch_self* and *smp_plug* devices are parts of the SMP implementation of MPI-BIP (GEOFFRAY; PRYLLI; TOURANCHEAU, 1999) and are used to provide a better performance when compared to the original operating system inter-process communication mechanism.

All the issues related to network heterogeneity are hidden by the Madeleine software layer as shown in the architecture presented in the figure 3.5.

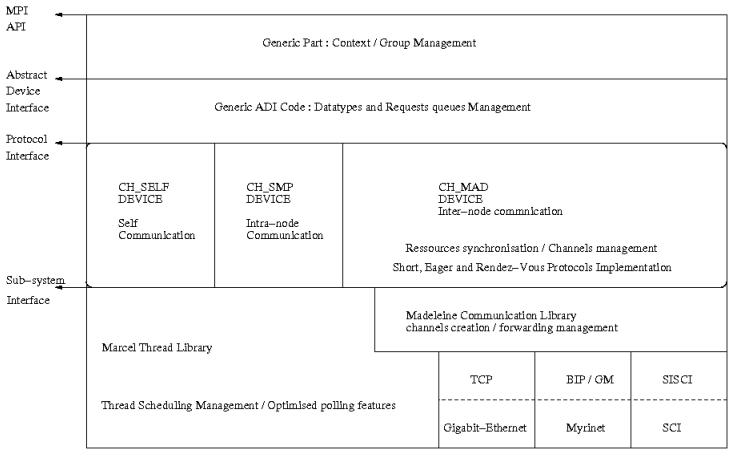

Figure 3.5: MPICH/Madeleine Architecture

3.3.3 Main Features

3.3.3.0.9 Multi-protocol Communication

The multi-protocol communication features included by the MPICH/Madeleine enables the communication involving two different protocols, for instance when communicating two processes in different cluster. A set of high performance networks interconnection like Myrinet, Giganet, SCI, Fibre Channel and others are available.

One of the main benefits of the multi-protocol capabilities is the possibility of using high performance networks on collective operations, even when all the process are not capable of using the same protocol.

3.3.3.0.10 Automatic Forwarding Mechanism

In order to enable communication between nodes that do not have a direct connection, an automatic forwarding mechanism is offered in the version 3 of the MPICH/Madeleine tool (AUMAGE; MERCIER, 2003). This mechanism creates the possibility of communication between nodes that does not have a direct physical link.

This mechanism depends on a manual configuration file defining the topology, the available protocols and forwarding nodes. Also the advantages of the multi-protocol communication may be present, depending on the availability and configuration.

The forwarders are special nodes located in gateways, that don't have an accessible rank and just serve the purpose of forwarding messages.

3.4 H_2O MPI

The H_2O MPI (KURZYNIEC; HWANG; SUNDERAM, 2005), developed in the Distributed Computer Lab of the Emory University intends to facilitate and simplify the execution of MPI programs across multidomain clusters. It leverages the H_2O distributed computing framework to route MPI messages across heterogeneous clusters located in different administrative or network domains.

Its approach involves the instantiation of customizable agents (*pluglets* in the H_2O taxonomy) at selected locations. These *pluglets* serve as proxies that relay messages between individual domains as appropriate, transparently performing address and other translations that may be necessary.

This project follows two main guidelines:

- develop a comprehensive support to heterogeneous machines (hardware and operating system) and interconnection network, offering operation across firewalls and failure resilience;

- leverage the component architecture of H_2O to provide value added features like dynamic staging, updating of proxy modules and selective, streamlined functionality as appropriate to the given situation.

H_2O MPI intends to provide a smooth transition for cluster applications to be executed in grid environments. For this reason no significant modifications to already existing MPI codes are necessary. Resources allocation and management, deployment and security are out of the scope of the project and must me treated apart.

The following subsections present the main H_2O MPI building blocks, its architecture and also the main features included on this tool.

3.4.1 Building Blocks

The H_2O MPI is built on the top of the FT-MPI (FAGG; BUKOVSKY; DONGARRA, 2001) and with the support of the H_2O framework on crossing-firewall configurations.

3.4.1.0.11 The FT-MPI

The H_2O MPI (KURZYNIEC; HWANG; SUNDERAM, 2005) is originally implemented on top of MPICH, because it is one of the most widespread machine-independent implementation. But the support to the FT-MPI (also based in the MPICH implementation) is considered of importance to deal with resources failures and dynamic offer of resources.

Different of the MPICH/Madeleine and the MPICH-G2, the H_2O is also not an implementation of the ADI, but an infrastructure that stands on top of MPI. According to the user needs, the fault tolerance mechanisms can be enabled, so offering a more reliable environment; obviously, at the cost of a degradation on the performance that according to the authors is really small.

3.4.1.0.12 The H_2O framework

Also developed in the Emory University, H2O is a middleware platform for building and deploying distributed applications. Conceptually, H2O is a distributed component Java-based framework, developed according to the CCA (Commom Component Architecture).

The main difference between H2O and other component frameworks, such as J2EE or Globus GTK, is that it removes the static binding between service deployer and resource provider. That is, H2O allows not just container owners but any authorized third parties or clients themselves to deploy services into the container.An usage scenario to this concept are MPI applications that can take profit of it to deploy forwarders in specific points of a grid infrastructure. Hence, resource sharing can be automatically achieved.

H2O also features a simple APIs for remote component deployment and management, and inter-component communication. H2O components can communicate via remote method invocations (both synchronous and asynchronous), and through a publisher-subscriber distributed event model. The communication layer offers a selection of messaging protocols (including JRMP, SOAP, RPC) and customizable transport stacks (SSL, compressed sockets, JXTA sockets, single-port tunneling, in-process sockets, all of which can be mixed in many combinations). These protocols can be configured so that multi domain firewalled environments could be used to execute large MPI applications.

3.4.2 Architecture

The architecture presented by the H_2O MPI is made of *pluglets* within specific H_2O kernels, located in strategic points of the infrastructure (e.g. frontends) and H_2O proxy pluglets in each of the resources involved in the computation. The figure 3.6 depicts these two main elements and their connection in a point-to-point communication between two resources with a firewall between them.

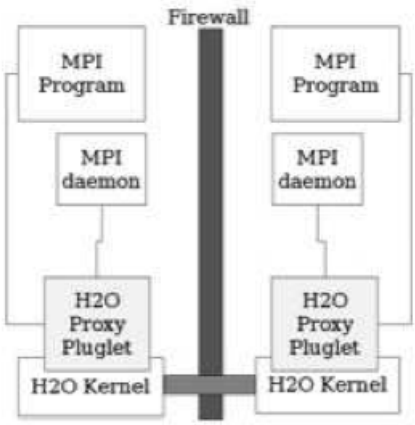

Figure 3.6: H_2O MPI Architecture: simple example

According to (KURZYNIEC; HWANG; SUNDERAM, 2005), some optimizations related to the communication between pluglets were in progress. These enhancements would guarantee a better performance not just in point-to-point communication but also in collective communication. However, more information about these enhancements were not provided yet.

3.4.3 Main Features

3.4.3.0.13 Message Tunneling

Due to the fact that H_2O uses well-known port numbers and that is capable of tunneling communication via HTTP, it is possible to configure firewalls appropriately to allow forwarding.

Tunnels are dynamically created and used as needed, being completely transparent to MPI applications. The cross-firewall communication is accomplished by means of some modifications introduced in the MPICH2 library. Because of these modifications, all messages are re-routed through an H_2O pluglet rather than directly to the remote MPI process. By doing so, the *pluglet* takes control of the communication and creates properly the best channel to connect two ranks (eventually a direct connection between MPI deamons).

3.4.3.0.14 Asynchrony on communication process

As performance is of crucial importance in most MPI applications, any proxy-based solution must incur as little overhead as possible. Some degradation of performance is inevitable because communication must be forwarded by the *pluglets*.

Some mechanisms are included in order to reduce the impact of this indirection: the existence of additional threads responsible for message transition and pre-configuration of channels and tunnels are some of them. As a result, according to (KURZYNIEC; HWANG; SUNDERAM, 2005) , the overhead can be greatly reduced.

3.5 Chapter Conclusion

This chapter presented some of the main tools which aim at enabling the usage of MPI in computational grids. As discussed in this chapter, these tools are based on a variety of approaches, from solutions that implement a multi-protocol communication layer (PACX-MPI and MPICH-Madeleine) to solutions entirely based on grid middlewares such as Globus (MPICH-G2) and message forwarding mechanisms (H_2O). However, they share a common goal: to support unmodified MPI applications.

In general, this approach can be considered appropriate to cope with legacy applications. However, it lacks of mechanisms to adapt existing applications, usually conceived to clusters, to a different environments. The result of experiments (present in the section 7.3.2) as well as previous works (KELLER et al., 2003) have proved that this is an step indispensable to maintain application performance and scalability. Besides, in general, traditional MPI applications are quite static in design and this issue must be treated either on application level or through a middleware capable of taking changes on the environment into account.

In next section, we present our approach that intends not just to offer support to unmodified applications but also a specific API to adapt existing applications to a hierarchical environment. The evaluation also shows the clear advantage that a hierarchical implementation can offer in relation to flat algorithms. A more direct comparison among these tools and the presented work is presented in the section 7.4.

4 PROPOSAL FOR GRID-AWARE HIERARCHICAL PROGRAMMING

This chapter presents one of the main contribution of this work: a proposal for grid-aware hierarchical programming. The goal of this chapter is to present the specification of the work, completely independent of the underlying support and implementation details. Even if some of the decisions are motivated by the availability of some tools, this proposal intends to present the user interface exempted from the underlying support, so that these mechanisms could be implemented differently or extended on different contexts.

To begin, we expose the main reasons that lead us to take this approach. After, we present a simplified API specification as well as some pertaining implementation issues and details. This Chapter also discusses a practical way to design hierarchical applications or adapt existing ones by using the proposed API to take into account available resources.

4.1 Rationale

When talking about high performance computing, communication has ever been the most expensive (in terms of time-consumption) part of applications, followed by memory access. With the constant improvement of processors performance (Moore's law), the gap between processor and network speeds is increasing even more.

We consider in this work that grids are hierarchical by nature: *i.e.* a grid can be considered as a set of multi-core nodes regrouped on multi-processors PCs, organized within clusters, then interconnected through wide-area networks. Considering that this hierarchical network organization also leads to different network performances (latency/bandwidth), programmers must be encouraged to give preference to communications between MPI processes that are neighbors in the cluster instead of communications between MPI processes lying onto different clusters of the grid. Worthy of notice that this has been proved to be a good approach to improve applications performance (DONG; KARONIS; KARNIADAKIS, 2006).

As seen in the section 2.2, MPI became a standard in high performance computing, and for this reason, the usage of this interface to grid programming is an interesting research nowadays. Even if cluster-based grids have a hierarchical organization, the idea of hierarchical communication does not exists at all in the standard. For this reason, we believe that an extension to the standard would enable it to grid computing and, at same time, facilitate grid platforms to be used for the execution of high performance applications.

More precisely, achieving our objective requires: an intuitive abstraction to deal with a hierarchical infrastructure, a framework capable of deploying the application on a grid infrastructure, an API to access such features and provide programmers with topology information and, obviously, a support for inter-cluster communication and process management. Next sections present in more details each of these requirements.

4.2 Specification

As previously presented, we intend to keep as much, as possible, the MPI programming style. In order to do so, we propose some extensions to already existing MPI abstractions, such as communicators and process rankings, to include support for hierarchical communication and topology discovery, having the grid complexities hidden by the framework.

4.2.1 New MPI communicators

Ideally, we consider a grid as a layered architecture:

- the first and lowest level (L1, for simplicity) is characterized by each processing unity of architecture of a node: single versus multiple-processor systems, single versus multiple-core systems and memory structure (SMP, NUMA, etc.).
- the second level (L2) is the representation of a computing node which potentially include multiple L1 entities;
- the third level (L3) consists of the set of processing nodes defining a cluster. Each grid node aggregated a number of processing nodes between a few dozens to a thousand nodes, usually interconnected with a fast network;
- the fourth level (L4) consists of the set of clusters defining a grid node. These clusters are typically interconnected by fast links. The main characteristic here is that resources are geographically close to each other (in relation to other resources on the grid);
- the fifth level (L5) represents the grid, which consists of a small number ($<$ 10) of geographical sites (grid nodes). These grid nodes are interconnected by a wide area network (WAN), and so, we can expect majors delays and limited bandwidth on the interconnection;

As the operating system already addresses the first level and the standard MPI the second level, we introduce two new communicators to deal with the fourth and fifth levels:

- `MPI_COMM_SITE` communicator: contains references to all the allocated nodes within a given site of the grid;
- `MPI_COMM_GRID` communicator: contains references to all the nodes in the whole grid.

From now on, we refer to these two communicators as synthetic communicators in comparison to natural communicators. In the first prototype, the high-level

communicators are static structures offered as an runtime abstraction. For this reason, there is no support to operations over communicators like `MPI_Comm_Split`, `MPI_Comm_Merge` and `MPI_Comm_Dup`. For now, MPI topologies are also not supported at site and grid levels.

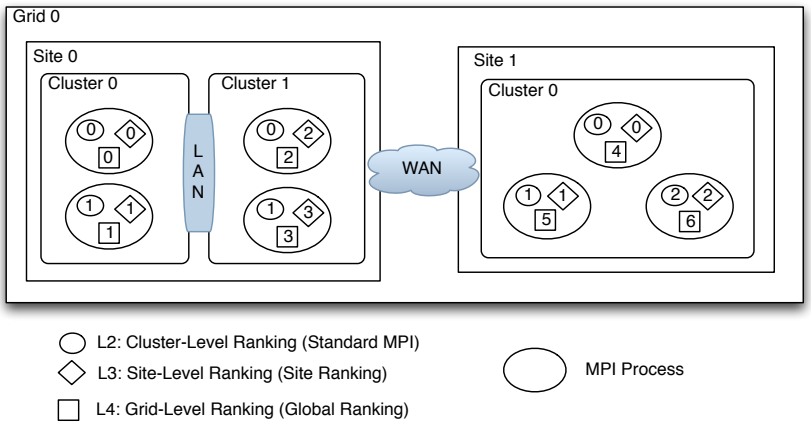

Figure 4.1: Hierarchical Communicators and Ranks

The figure A.3 shows how the ranks are organized in each of the communicators. This is a pretty simple and adequate way of giving ranks to MPI processes in a hierarchical environment. Simple because it can be easily done during the startup of the environment, so having an unified global view. Adequate as it enables an easy and non-error prone way to identify processes, indeed keeping the communicator abstraction.

4.2.2 New primitives

In addition to the abstraction of hierarchical communicators, we include a set of primitives, mainly addressing topology discovery. Just to cite some examples:

- the `MPI_ClusterInfo` and `MPI_SiteInfo` data structures contain information pertaining to a given cluster or site respectively, the primitives `MPI_Comm_getMyClusterInfo (MPI_Comm comm, MPI_ClusterInfo *)` and `MPI_Comm_getMySiteInfo (MPI_Comm comm, MPI_SiteInfo *)` are used to get the information of the cluster and site of the process that calls these primitives, and `MPI_Comm_getAllClusterInfo(...)` and `MPI_Comm_getAllSiteInfo(...)`, that returns information about the entire topology;

- `MPI_COMM_SITE_GATEWAYS` and `MPI_COMM_GRID_GATEWAYS` are special communicator that aggregates one node (usually the rank 0) of each cluster and site within a site. The intention of this communicator is to offer an abstraction to induce an explicit hierarchical programming, and can be used in order to reduce the many indirections (and performance degradation) of global communications.

- `MPI_Comm_getNeighborhood (int rank1, int rank2, MPI_Comm)` that can be used to retrieve the neighborhood relationship between two processes of a given communicator, *i.e.* how many hops we can expect for a message from the `rank1` process to reach the `rank2` process of a given communicator, given the current resources organization;

- `MPI_Comm_translate(int in_rank, MPI_Comm comm_in, int* out_rank, MPI_Comm comm_out)`: as we introduced new communicators, it may happen that a given process has different ranks depending on the communicator context considered (see an example on the table 6.1). So, this primitives enable users to translate ranks between communicators.

Some other standard MPI primitives still functional with the new communicators, such as `MPI_Comm_rank` and `MPI_Comm_size` . However, at least in the first specification, the intention is not provide full support to all the already existing primitives, when used with the new communicators. In fact, just a minimum set of the most used original primitives must be initially supported. For the communication process, the supported primitives are: `MPI_Send`, `MPI_ISend`, `MPI_Recv`, `MPI_IRecv`, `MPI_Bcast`, `MPI_Barrier` `MPI_Scatter`, `MPI_Scatterv`, `MPI_Gather` and `MPI_Gatherv`.

4.2.3 Communication Semantics

A valid MPI implementation guarantees certain general properties:

- Order of point-to-point messages are non-overtaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives at the application level. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard `MPI_ANY_SOURCE` is not used in receives.

- Collective operations always happen on a communicator context: All the MPI collective operations happen in the context of a communicator and all the process of the given communicator must participate, independing on the the nature of the communication. Obviously, depending on the collective communication either all the process perform the same operation (e.g `MPI_Barrier`) or the primitive implies on a different behavior of a coordinator (e.g the root of an `MPI_Bcast` operation).

- Collective operations on a given context are FIFO ordered and isolated from each-other (no interleaving): the implementation must also take care to avoid a false match between similar operations on different contexts, for instance if a set of process perform an `MPI_Gather` on `MPI_COMM_SITE`, followed by an `MPI_Gather` on `MPI_COMM_GRID`. Even if this property is not defined on the MPI standard, it is a necessary condition to avoid deadlocks.

These properties must be taken into account on the implementation of the communication operations, and this may imply on the creation of message queues dedicated for each new synthetic communicator context.

4.3 Support to unmodified applications

In previous sections, we have discussed the need of hierarchical structuring of parallel applications to keep scalability by taking into account grid resources topol-

ogy. This need is specially important for highly communicating application as performance degradation is closely related to differences in performance from local to distant networks.

Differently, some applications, such as embarrassingly parallel ones, do not require a big amount of communications among process. In general, these application scale well and take profit of grid resources, even with a flat organization. Considering that a hierarchical solution implies in a more complex development, sometimes it makes perfect sense to run unmodified applications on grid resources. Besides, some users might be interested on running their applications without the pain of adapting the data distribution and communication process.

For such cases, a C macro (#UNMODIFIED) is defined. This macro can be added to any MPI source codes and when it is present, the natural communicator `MPI_COMM_WORLD` will have the same behavior than the synthetic `MPI_COMM_GRID`.

4.4 Chapter Conclusion

This chapter presented a proposal for hierarchical grid-aware communications along with an API that extends the MPI standard to offer programmers a way to express grid-aware algorithms from scratch or through modifications in existing applications. These definitions were made to be general, so that they could be implemented through different technologies. In the section 6, we present an implementation based on components, but a pure MPI implementation could be also easily conceived.

These extensions follow the MPI standard so that developers used to the MPI standard could easily use them. Besides, some new primitives intend to offer information about the executing environment, which is an important feature to develop applications adaptable to different environments. As we will see later, in the section 7, this might not an easy task. Nonetheless, applications can really benefit of grid resources and scale more if they follow a hierarchical approach.

The idea of extending the MPI standard is neither compete with the standard or replace it, but rather offer an alternative approach that completes the MPI standard offering some simple abstractions to fulfill grid requirements, addressing the main issues. This is an ambitious goal that depends not just of the extensions, but also of their implementation and runtime. The section 6 presents a first prototype that implements this interface. Some code examples and a qualitative evaluation of this proposal can be seen in the chapter 7.

5 DEVELOPMENT RESOURCES

This chapter presents the software resources that were used on the development of the proposal described in the previous chapter. We also present, in this chapter, some important concepts that may help to understand the development of the runtime (Chapter 6), namely the deployment and component model.

Initially, we present the ProActive grid middleware (BADUEL et al., 2006). So, we present the ProActive deployment framework, the already existing integration between Java and native code (MPI). The last section brings the Grid Component Model (GCM) and the main concepts and features that were used to develop this work.

5.1 The ProActive Grid Middleware

The ProActive middleware (BADUEL et al., 2006) is a 100% Java middleware, which aims to achieve seamless programming for concurrent, parallel, distributed and mobile computing. The base model is a uniform active object programming model. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls. Active objects are remotely accessible via asynchronous method invocation. This is provided by automatic future objects as a result of remote methods calls, and synchronization is handled by a mechanism known as wait-by-necessity.

Besides of its own programming model, ProActive features: group communication with dynamic group management, an object oriented SPMD model (OOSPMD), fault tolerance with checkpointing, a powerful deployment model based on XML descriptors that offer support to numerous network protocols, cluster resource managers and grid tools as well as a grid component programming model (GCM) based on the Fractal specification (BRUNETON; COUPAYE; STEFANI, 2004).

Specially relevant to this work are: the deployment framework, the reference implementation of the ProActive/GCM and the MPI Code Wrapping Mechanism. The next subsections present these features in more details.

5.2 ProActive Deployment Framework

The ProActive middleware includes a powerful deployment model and framework (BADUEL et al., 2006). A first principle is to fully eliminate from the source code the following elements: machine names, creation protocols and registry/lookup protocols. The goal is providing users with the capability to deploy any application

anywhere, without changing the source code. For instance, one must be able to deploy grid MPI applications using various protocols, rsh, ssh, Globus, LSF, etc., for the creation of the JVMs needed by the runtime or the MPI applications themselves. In the same manner, the discovery of existing resources can be done with various protocols such as RMIRegistry, Jini, Globus, LDAP, UDDI, etc. Therefore, the creation, registration and discovery of resources is done externally to the application.

A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. The description should indicate the various parallel or distributed entities in the program or in the component.

These two main principles rely on the notion of *Processes* and *Virtual Nodes (VNs)*. *Processes* are defined by the protocol or set of protocols necessary to allocate resources and launch an application as well as all the configuration needed on this process (logins, passwords, keys, resources requirements, entry points, etc.). After using the right protocols, the resources can be acquired through the abstraction of *Virtual Nodes*, which are potentially composed by many *Nodes*.

The VNs are described in XML descriptors and defined by three different aspects: the mapping of VNs to Nodes and to JVMs, and the *process* to launch these VNs, i.e. the way to create or to acquire JVMs and how to register or to lookup VNs. After the activation of a VN, the underlying infrastructure is launched and the application can use the resources through references, called *Nodes*.

Some other aspects can be defined in the deployment, such as security, tunneling of communications and fault tolerance. Also, the support to file transfer mechanism plays an important role as it is responsible for transferring binaries and input data to the resources that will run the MPI application.

The usage of the ProActive deployment, as well as the transfer of files (binaries and input/output data) is explained in details on the section 6.3.

5.3 ProActive MPI Code Wrapping

The Code Wrapping Mechanism, recently introduced in the ProActive middleware proposes a simple wrapping method designed to easily launch MPI applications on clusters or desktop grids. Besides, an API enables the coupling of several codes, MPI and/or Java.

Two kinds of applications may be interested by the code wrapping mechanism: first, unmodified legacy MPI applications that intends to run onto a single cluster and second, the development of conventional stand-alone Java applications using pieces of MPI legacy codes.

The main features of the wrapping mechanism include:

- Transparent wrapping and deployment of MPI applications: consists in wrapping MPI processes within ProActive active objects, adding capabilities for deployment, control and communication. Due to the use of the ProActive deployment descriptors, a MPI application can be deployed using a large number of protocols and schedulers. In addition, the ProActive file transfer mechanism enables the transfer of application binaries and input data;

- Support for "MPI to/from Java" point-to-point communication: a set of MPI-like C functions and Java classes permit the point-to-point exchange of messages between the two worlds. Furthermore, this feature can be useful to

enable the communicating between two MPI processes potentially located at different domains under firewalls or even private IPs, through a ProActive layer;

- Control of MPI processes: through the use of the code wrapping, it is possible trigger job execution, to kill MPI processes and retrieve execution status and results from the native MPI processes;

5.4 The Grid Component Model (GCM) and the ProActive/GCM Reference Implementation

The Grid Component Model (GCM) (GRID COMPONENT MODEL SPECIFICATION, 2007), defined by the Institute on programming models of the EU CoreGRID project, defines a lightweight component model (the GCM) for the design, implementation and execution of grid applications. The key problematic addressed by the GCM are programmability, interoperability, code reuse and efficiency.

This model relies on the Fractal component model as a basis for its specification. In fact, GCM can be considered an extension to the Fractal specification, addressing grid requirements like deployment and collective communication. In this sense, the ProActive/GCM (MOREL, 2006) is a reference implementation of the GCM which provides a component framework that aims at fulfilling the needs of grid programming.

Our research investigates how to use a component model (in this case the GCM), to support MPI over a grid. This choice is motivated by some of the characteristics of the GCM and extensions that are available thanks to its implementation in ProActive. Some of them are:

- Hierarchical structure and adaptable composition: grids are inherently hierarchical and, in general, composed by a large set of heterogeneous resources. As the GCM enables the composition of components in a hierarchical way through the encapsulation of primitive components within composites, it offers a straightforward way to model a grid infrastructure into a component framework. Besides, the capability of creating and re-arranging bindings on-the-fly can be very useful to express in the component level the interactions of distributed environments, such as MPI.

- Separation of concerns: the concept of separation of functional aspects and non-functional ones is very interesting for any distributed systems, as it helps to separate the control of resources and binding from the communication that happens in the application level. so, from a developer point-of-view, the intrinsic characteristics of the environment and topology, that reflect into component bindings, can be completely hidden.

- Encapsulation of code: components provide built-in capability of encapsulating both, native and Java code. So, we can provide a common component interface to MPI processes running in heterogeneous resources, possibly in different domains. By wrapping code, we can aim at obtaining a performance as good as native MPI implementations in intra-cluster communication.

- The collective interfaces: GCM collective interfaces are of different natures. These collective compositions of interfaces enable the externalization, at the component level, of the parallel nature of the component code, which is the case if it uses MPI. More precisely, collective interfaces feature built-in possibility for collective synchronization and more complex communication operation: multicast, gathercast and an optimized way to combine a gathercast plus a multicast interfaces for binding two hierarchical components composed of a different number of similar components, yielding to a MxN collective interface.

These concepts are important at different moments of the conception of the runtime. Initially, a hierarchical structure permits to create a component model similar to the topology of available resources. The encapsulation of codes permits an easy integration of MPI codes and collective interfaces ease the development of a high-level inter-cluster collective communication process. The idea of separation of concerns is also very useful, as it permits to handle grid aspects at the non-functional level.

5.5 Chapter Conclusion

In this chapter, we have presented the tools that were used to develop the prototype that supports the work proposed on this document. In practice, this implementation can be separated on different functionalities:

- deployment of the component-based runtime and MPI applications, that was implemented due to some small extensions to the ProActive deployment framework;
- external control of MPI applications and native to/from Java communication that takes profit of the features presented by the MPI code wrapping mechanism;
- a hierarchical component-based framework implemented thanks to the GCM components and
- support to inter-cluster and collective communication, offered by the core library of the ProActive middleware.

The next Chapter explain in details the definition and implementation of each of these mechanisms.

6 DESIGN AND IMPLEMENTATION OF THE COMPONENT-BASED RUNTIME

In the Chapter 4, we have presented some extensions to the MPI standard that were conceived to ease the development of grid-aware algorithms. The support to these primitives requires a set of functionalities on the middleware side:

- deployment of applications on a grid infrastructure;
- distribution of input data and binaries on the grid infrastructure;
- support to primitives that offer resources and environment information;
- handling of inter-cluster communications.

Initially, this chapter presents some principles taken into account to the design and development of the component-based runtime. Then, we present the component-based infrastructure, how the deployment happens, the mapping of this infrastructure to grid resources and how inter-cluster communication takes place within this infrastructure as well as some important optimizations on the communication process. The last subsection presents some conclusions around the development of the prototype.

6.1 Principles

A set of principles have guided the design and implementation of the component-based runtime. These principles may help understanding some of the choices done on the design and implementation of the prototype. The next paragraphs depict each of the general principles.

1. Use native MPI communication whenever possible. Despite of the included optimizations, communication between components is slower than the communication between MPI processes. This comes from the fact that component communication relies on asynchronous Java remote method invocations that depend on the serialization/deserialization of objects and queuing of messages on the JVM. Besides, the main MPI implementations present support to high performance communication protocols (e.g. InfiniBand, Myrinet, Quadrics, etc.), while Java communication depends on expensive plain TCP/IP messages. For this reason, the runtime must be able to perform communication through MPI whenever a direct communication between processes is available.

2. Use direct communication whenever possible. Due to the usage of components to model a hierarchical grid infrastructure, the communication between two end-points sometimes imply on many indirections, even if a direct physical link between processes is available. In order to soften the impact of these indirections, the runtime must be capable of identifying the shortest path between two processes (and wrapping components) to perform the communication.

3. Hierarchical treatment of collective communications. In general, point-to-point communication is negatively affected by the hierarchical approach as it depends on indirections and intermediary copies. Differently, collective communications may take profit of the hierarchy to improve performance. As this kind of communication depends on the participation of many entities, when it takes place on a multi-cluster context, it can be decoupled according to the operation and made in parallel on each cluster before the communication with other clusters, so reducing the amount of expensive inter-cluster communication and sequential data treatment.

4. Essential grid issues as hidden from users. As a matter of fact, the main idea behind grid middlewares is to hide grid complexity from the users, so that users could concentrate on the development of applications rather than solving grid issues. The design of this prototype may not be different and all the issues related to heterogeneity, distribution, communication and execution are handled transparently. In addition to ease the usage of the framework, the environment must give a better control of the environment and help improving performance. The separation of concerns inherent to component oriented programming may help addressing this principle.

6.2 Software Architecture

The support to the primitives presented in the chapter 4 is based on a layered architecture organized according to the figure A.4. This stack is a representation of the software layers present on each computing node.

In the top, we have a grid-aware application, that is interfaced with the entire framework through the MPI standard and the extensions presented in the section 4.2. At the interface level happens the control whether the communication will go through the native MPI implementation (in general, intra-cluster communication) or through the components and the underlying JVM (in general, inter-cluster communication). And, at the bottom, there is the operating system (OS) that takes care of process management and network.

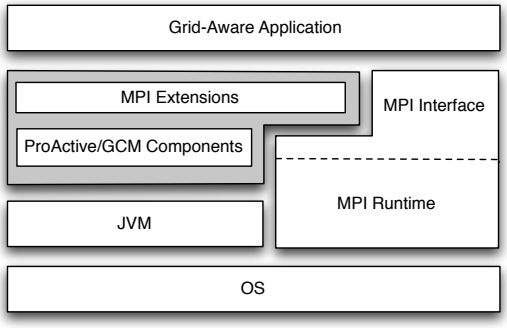

Figure 6.1: Architecture of the Framework

The gray box delimits what is actually proposed and develop in the context of this work.

6.3 Runtime Deployment and Data Distributions

The deployment of a *gridified* MPI application is mainly based in the ProActive deployment (section 5.2). It encompasses allocation and access to resources, deployment of the MPI application and component framework. The idea is to hide from users all the issues related to resources management (principle 4).

As already discussed in the section 5.2, the ProActive deployment mechanism is based on the concept of *processes* that are mapped into the *virtual nodes* abstraction. However, an extension of this mechanism was necessary to support the deployment of multiple MPI applications and simultaneously the deployment of the component infrastructure.

This extension introduces on the ProActive deployment the concept of dependency between processes. The idea behind the deployment dependencies is to enforce an order of the deployment of applications. In the case of this work, the components will wrap running MPI processes once they are deployed and, for this reason, they depend on the successful deployment of MPI applications.

The following XML snippet shows, in practice how a simple dependency looks like on a deployment descriptor:

```
<!-- dependent process -->
<processDefinition id="dpsCPI">
    <dependentProcessSequence
        class=
        "org.objectweb.proactive.core.process.DependentListProcess">
        <processReference refid="mpiCPI" />
        <processReference refid="sshProcessList" />
    </dependentProcessSequence>
</processDefinition>
```

After defining the dependency, it is just a matter of specifying the required process (an MPI process on this example) and the dependent one (SSH on this example).

Here, the required process is an MPI process, which launches the binary *"proactive_poisson3D_cubic"*, with the mpi command defined by the variable *${MPIRUN_PATH}* on 4 machines, passing a set of parameters *${P3D_PARAM}* to the application:

```
<!-- mpi Process -->
<processDefinition id="mpiCPI">
    <mpiProcess
          class=
          "org.objectweb.proactive.core.process.mpi.MPIDependentProcess"
              mpiFileName="proactive_poisson3D_cubic"
              mpiCommandOptions="\${P3D_PARAM}">
        <commandPath value="\${MPIRUN_PATH}" />
        <mpiOptions>
```

```
            <processNumber>4</processNumber>
            <localRelativePath>
                <absolutePath value="\${POISSON_3D_FOLDER}"/>
            </localRelativePath>
        </mpiOptions>
    </mpiProcess>
</processDefinition>
```

And the dependent process defining the deployment of a Virtual Node composed by 4 nodes, deployed using the SSH protocol.

```
<!-- ssh process -->
<processDefinition id="sshProcessList">
    <processListbyHost
    class="org.objectweb.proactive.core.process.ssh.SSHProcessList"
                 hostlist="node1 node2 node3 node4">
        <processReference refid="localjvm" />
    </processListbyHost>
</processDefinition>
```

The appendix B shows XML Schema Definition (XSD) that defines a dependent process.

In practice, the deployment of a *gridified* application happens according to the following steps:

1. Allocation of resources (if needed);
2. Automatic generation of deployment descriptors;
3. Deployment of pure MPI applications, one instance by cluster;
4. Deployment of the component architecture, on the same set of nodes;
5. Begin of handshake between MPI processes and wrapping component with loading of native libraries and exchange of identifiers;
6. Definition of bindings among the components, accordingly to underlying MPI infrastructure;
7. End of handshake and start of the application execution.

The handshake phase works as a barrier that synchronizes all the native MPI processes and components. An important optimization on the component happens in this phase and is explained in the section 6.6.1. Once the application finishes, a new handshake takes place to finalize the runtime execution.

Besides of the deployment of the runtime, *gridified* applications usually have need of input data, application binaries and libraries. Together with the definition of the different processes schemes, the ProActive deployment also includes support to a file-transfer mechanism. For now, the file-transfer must be configured by hand and includes the transfer of binaries if they are not available on computing nodes.

(possibly multiple compilations for the different architectures). The file-transfer mechanism is specially useful when resources does not share a filesystem through NFS, for instance. Even if most clusters present NFS nowadays, this is not the case when coupling resources on different clusters/organizations.

6.4 The Component Infrastructure

The utmost goal of the runtime is to offer efficient and transparent inter-cluster communication. This is achieved by means of an overlay infrastructure of components modeling the hierarchy of resources allocated to the execution of a given application. This infrastructure does not present any central entity coordinating the global execution and only exists in runtime, with no need of daemons or software installation other than an MPI distribution.

The next subsections present the main components involved on the development of this infrastructure, how these components are mapped to reflect resources topology as well as their interconnections to provide connectivity of MPI process located on different clusters.

6.4.1 Definition of Basic Components

Two different mechanisms are necessary to offer inter-cluster communication: (i) communication between native MPI process and the Java environment and vice versa and (ii) routing of messages throughout the network. Each of these features is offered by a different type of component:

- a **wrapper** component (Figure A.5.a) is the most elementary kind of element used in the component infrastructure. It wraps an MPI process and is responsible for the MPI to/from Java communication. As such, it is also in charge of the encapsulation of the MPI messages into objects, including information useful to route messages within the component infrastructure (section A.11) and by the serialization/deserialization of messages.

  The wrapper component presents just a server (drawn on the left hand side of a component) *Srv* interface and a client (drawn on the right hand side of a component) *Clnt* interface, that are bound to a **clustering** component which represents the cluster where the MPI process (and so the component) is located.

- a **clustering** component (Figure A.5.b) is a generic component capable of clustering lower-level components. In our case, a **clustering** component at level 3 (L3) groups L2 (**wrapper**) components, and the **clustering** component at level 4 (i.e. at the level of sites in the grid hierarchy) groups L3 (**clustering**) components. The same for a component at level 5 (a grid) that groups L4 components (sites). These components are responsible for communications between different clusters and sites of the grid.

  The clustering components present a *Srv* server interface that is responsible for receiving requests coming from both lower-level or higher-level components. Even if the *Srv* interface is not a gathercast interface (as defined in (BAUDE et al., 2005)), it can perform gather operations of the received messages within

the implementation of the component by the interception and analysis of incoming messages.

As client interfaces, the **clustering** components present 2 multicast interfaces: one for sending messages to upper level **clustering** components (*GoUP* interface) and other for lower-level components (*GoDown* interface). These multicast interfaces make capable to the **clustering** components to communicate with multiple components in parallel and is used, in practice, to implement most of MPI collective primitives. The semantic adopted by these interfaces depend on the type of communication and is explained in details in the section A.11.

Figure 6.2: a) Wrapper component b) Clustering component

The next two sections present how these components and interfaces are assembled in practice to form a grid infrastructure capable of handling inter-cluster communication.

6.4.2 Mapping Resources to a Component Infrastructure

As already discussed in previous sections, the component infrastructure is a model that represents the resources topology using the components described in the section A.10.1. The idea is to make possible to any process within a given cluster to send point-to-point and collective messages to any other process or set of process on the grid.

The definition of the component organization starts by the first phase of the deployment when nodes are allocated or simply specified by the user. On this infrastructure, all nodes and corresponding *wrapping* components receive an unique identifier, formed at the lowest level of the hierarchy by the identifier of the site, cluster and rank of the native process within the most embedding communicator in which it runs, i.e. in the one corresponding to the cluster. The figure A.6 shows a simple deployment on two sites, with two and one cluster respectively and the total of 7 computing nodes. Each level defined in the infrastructure imply on the addition of one more identifier, being the top level composed by a single component that represents the entire environment and the bottom by processing nodes.

The figure A.7 shows the components responsible for the wrapping and inter-cluster communication. For clarity, the figure just presents the components deployed on the left branch of the hypothetical grid deployment (site *'0'* of the figure A.6).

On this organization, there is one *wrapper* component (L2, in white) for each of the launched MPI processes, one L3 *clustering* component by cluster and one L4 component on the top of the site. In general, *clustering* components are deployed on the frontend of the clusters (by default, the node 0 or manually specified at the

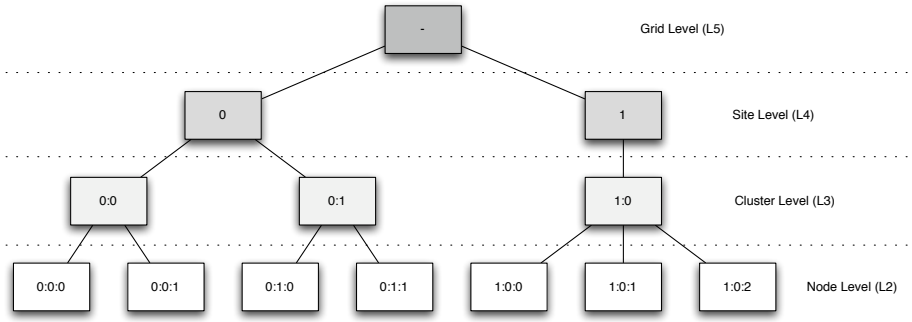

Figure 6.3: Resources/Components identification

deployment stage). If not defined otherwise, all the communication is handled by Java RMI, but other protocols can be easily defined (rmissh, ibis, http or even soap), when RMI communication is not allowed.

The figure A.7 shows the initial binding configuration, that obliges the usage of indirections through *clustering* components. The section 6.6.2 presents an optimization that softens this requirement on point-to-point communications, by following the principle 2 (section 6.1).

In the basic configuration, all the components have a single server interface (*'A'*) with the same behavior: messages are intercepted on this interface and depending on the message type, they are directly passed to the wrapped MPI process (point-to-point messages) or determine the components to wait for more messages (collective depending on more than once received message). This server interface is capable of receiving messages from any other component that references it. Besides of the inbound links on the *Srv* interface, *wrapper* components are interfaced with the external environment through a single client interface (*'B'*), bound to the component that represents the cluster where the component is deployed in.

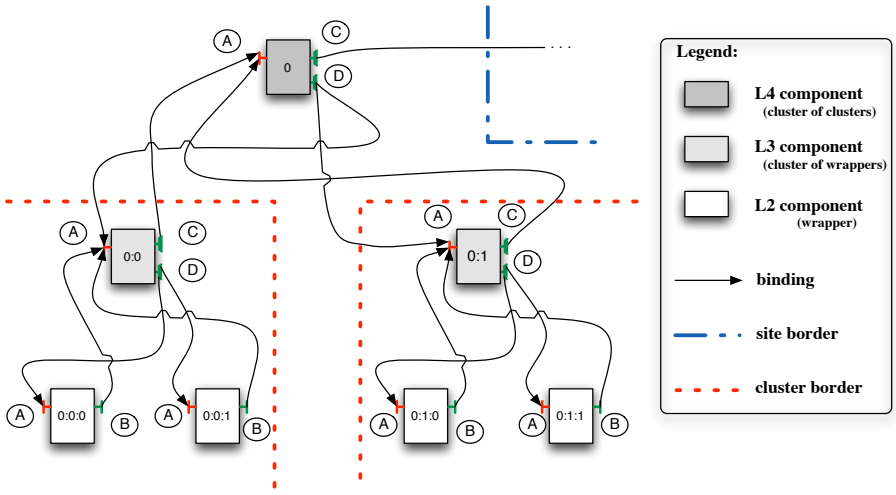

Figure 6.4: Assembly of Components

Clustering components (L3 and L4) are connected simultaneously to upper level components (interfaces *"C"*, connected to L4 and L5 components, respectively) and to lower level components (interfaces *"D"*, connected to L2 and L3 components, respectively). Even if the *GoUp* interface are only connected to the clustering com-

ponents, they are collective interfaces because they are collective interfaces because these interfaces can be connected directly to interfaces of the same level in order to crosscut levels in the components hierarchy(section 6.6.2).

6.5 Message Routing over the Grid

After launching the environment, a global view of the topology can be built and each process creates locally its own view of the topology, through a mapping between the communicators and unique identifiers. The following table shows an example of table (in this case, the table stored in the nodes 0:1:0 and 0:1:1 of the figure A.6).

Ranks \ Comms	MPI_COMM_CLUSTER	MPI_COMM_SITE	MPI_COMM_GRID
0	0:0:0	0:0:0	0:0:0
1	0:0:1	0:0:1	0:0:1
2	-	0:1:0	0:1:0
3	-	0:1:1	0:1:1
4	-	-	1:0:0
5	-	-	1:0:1
6	-	-	1:0:2

Table 6.1: Mapping of ranks within hierarchical communicators to component identifiers

Based on this table, the decision regarding whether messages will be sent through MPI or components can be done in the native process. This decision takes into account the type of message (point-to-point or collective), the communicator that gives context to the communication and the destination. The routing process has two distinct behaviors: that of the point-to-point communication and the one that happens in the context of collective communication, as explained below.

6.5.1 Point-to-Point Messages

Due to the existence of unique identifiers, the routing of point-to-point messages in the component hierarchy is straightforward. It just requires the sender process to get, from the conversion table, the identifier that corresponds to the destination and try to match the remote identifier with its own.

If the two first values of the identifier (site and cluster identifiers) are the same, which means the process are located on the same cluster, the sender turns the call into a simple `MPI_Send`, that is faster than sending messages through the component infrastructure, because it requires less intermediary copies. Indeed, MPI optimizations and support to high performance network devices and protocols can be used. Otherwise, it means that the source and destination are, at least, in different clusters and so the *wrapper* component will take care of the communication.

The communication handled by *wrappers* happens in two steps:

1. Get the recipient reference through its *GoUp* client interface. Supposing the source and destination are placed at different sites, this process happens again

to an upper-level *clustering* component, that represents a site in the topology. The section 6.6.1 presents an optimization of this mechanism;

2. After the reference to the destination component is obtained, it is possible to forward directly the message by triggering a ProActive communication between the two remote components that wrap the legacy MPI processes (taking profit of the equality of component and underlying active object references in the ProActive implementation the GCM model).

We could have preferred to explicit the bindings between wrapper components, and rely on such bindings for direct forwarding, which would have been much in line with component-orientation. Nevertheless, we consider this as a useful simplification for the run-time support, because it does not require to establish and maintain the bindings between any pair of wrapper components and usually not all pairs of MPI processes are going to communicate.

In the case of firewalled clusters or sites, where nodes do not have direct access to each other, instead of getting the reference of the remote component by following the component hierarchy, the messages themselves are forwarded through it. In this case, one extra copy for each hop of the path is required. Such feature makes use of the ProActive hierarchical deployment and forwarders.

In the receiver (in the extended API implementation), it is just necessary to check, according to the indicated communicator, if the message will come from a process in the same cluster, and perform a `MPI_Recv` or to expect a message from the enclosing component.

6.5.2 Collective Messages

As in standard MPI, the collective communication primitives take place within a given communicator context and all the ranks of this communicator must participate. According to the definition presented in the section 4.1, the hierarchical communicators are an abstract representation of a level in the hierarchy. Thus, the collective communication using the hierarchical communicators are, in fact, collective communications within a given level and so must be handled by the corresponding components.

The routing of collective communication is slightly more complex than the point-to-point. Despite the fact that point-to-point communications can be done directly between wrapper components, in the case of collective communication within synthetic communicators, it is mandatory to send them through the top-level components. The behavior of the communication depends on the kind of the collective communication, which may involve a root process (coordinator) or not.

For the sake of simplicity, we describe the collective communications done within a site. In the case of collective communication in the entire grid, the same happens, but twice, first from the L2 to the L3 clustering components and later from them to the L4.

- Broadcast: the processes involved in the broadcast of a message have two distinct behaviors. The one performed by the non-root wrapper components, that perform a receive as in the point-to-point communication and the behavior of the broadcaster (the root), that just includes the proper headers to the message and send it to higher levels by using the *GoUp* client interfaces, until it reaches

the right level. On its turn, the top-level representative component forwards the message to all the destinations through the *GoDown* client interface. The process of forwarding the message to multiple recipients is done in parallel due to the multicast nature of *GoDown*, bound to all the components in the level immediately above. The reception of the message at the broadcaster unblocks it. In scatter primitives, the behavior of the wrapper components is the same than the broadcast. However, instead of just forwarding the entire message to the recipients, the top-level components are in change of splitting the message in multiple ones before sending (in a similar way as MPI would do it).

- Gather: in the gather primitives, the behavior is exactly the opposite of the scatter. The root node performs a simple receive as in the point-to-point communication and non-root nodes perform a send to the top-level component of the given branch through the *GoUp* client interface of upper-level components. This top-level component on its turn, blocks until it receives all the awaited messages on its *Recv* server interface (one message from each of its sons). Then, this component is responsible for ordering properly the messages based on their header and merge them in a single message that is sent down the right branch of the hierarchy to the recipient. The Reduce primitive follows the same approach, but, instead of ordering, the *clustering* components make an operation over the data.

- Barrier: in the case of the barrier operations, the behavior of the nodes is that of the broadcast, i.e. send a void message to the higher-level components through the *GoUp* interface and wait for an acknowledgment. The higher-level component behavior is that of a gather to receive the messages and that of a broadcast to spread a message through the multicast *GoDown* interface, the messages that will unblock all the nodes involved in the barrier.

Each clustering component has a queue to store collective messages. The treatment of these messages follows a non-blocking FIFO (first in, first out) order and this enforces message ordering as done in most of the distributions that implement MPI standard to ensure correctness. The arrival of the first message of a collective call that depends on multiple calls (gather, reduce and barriers) creates a new entry on the queue and once all the needed messages are received the collective communication can proceed, which characterizes the non-blocking behavior.

6.6 Runtime Optimizations

This section presents three simple optimization in the communication process that enables to reduce the overhead on the usage of a component infrastructure. Initially, we analyze the caching mechanism that reduces the time needed to fetch components reference. Then, we analyze the tensioning technique which helps reducing indirections on a component infrastructure. The last mechanism in built-in on the implementation of the collective communications to make them more grid-aware.

6.6.1 Reference Caching and Pre-fetching

In the section A.11.1 we have seeing that the point-to-point communication depends on the obtention of remote references. This optimization is based on the fact

that process that already communicated with process located on different clusters are more likely to communicate again in the future.

The caching of references happens in two levels:

- at the *wrapper* components level: *wrapper* components keep remote references once the references are established. This avoids the re-fetching of existing references and lookup of remote components. The size of caches can be limited to reduce the number of opened sockets and memory consumption. By default, all the references are kept, until the maximum number of opened sockets is reached (in Unix, the maximum number of opened file descriptors). These references are stored on a hash, together with a timestamp and once the maximum number is exceeded, the oldest references are deleted, according to a least recently used (LRU) references replacement algorithm.

- at the *clustering* components level: in the components hierarchy, just the closest *clustering* component has the reference of *wrapper* components (in practice, a ProActive reference given by a URL where the remote object is bound). *clustering* components also keep references of all the references that were requested. This is useful because references that were obtained by one *wrapper* component can be retrieved by other components without the need of further indirections.

the pre-fetching of references is a complementary mechanism, which is used by default on the evaluations of this work (chapter 7), consist in creating a table of all the references of *wrapper* components (URL) and dispatch this table to all the clustering components. By doing this, the reference of a remote *wrapper* component can be retrieved with a simple message call.

6.6.2 Tensioning

At deployment time, *wrapper* components do not know the grid topology, i.e. they do not know if it is possible or not to communicate directly with other *wrapper* components in the grid. Actually, just the communication between the *wrapper* component and the lowest *clustering* component is ensured by the deployment. This is frequently the case when the deployment happens over multiple sites distributed geographically, but that offer connection between nodes relying on the same site, but on different clusters.

The tensioning mechanism is a technique commonly used in component-based software architectures (MOREL, 2006) that follows the principle 2 (section 6.1). It consists on a mechanism that remembers the shortest path between two software entities and establishes a minimal connection between them (direct if possible). The first invocation determines the shortcut path, then the following invocations will use this shortcut path.

The definition of the shortest path is, in fact, a try-error technique. Starting from the *wrapper* component that tries to contact directly the recipient. If it succeeds, a shortcut is created. If not, it transfer the control to the clustering component that tries to do the same. Once a component succeeds to contact the recipient (mandatory for the last clustering component, that is the father of the recipient component in the hierarchy), the path is stored on the involved *clustering* components and transmitted to the sender, that stores the entire path.

6.6.3 Hierarchical Collective Communication

In general, collective communication in the context of a given site or on the entire grid can benefit from the hierarchical approach to parallelize the execution of the call. Most of the implementations of the MPI standard present optimizations in the collective communications by means of trees of sockets when the communication involve more than a certain number of processes. When collective message calls arrive within a given cluster, they are handled by native MPI and can take profit of MPI optimizations but a further optimization is possible in some cases.

The following primitives include a specific treatment at the *clustering* components level:

- Broadcast: the optimization is pretty straightforward. It just consists on sending a single message to the clustering components that will effectively broadcast the messages internally;

- Scatter: the optimization is also intuitive and data is scattered in phases. Initially, data is scattered to send to *clustering* components just data that will be scattered in their context and this reduces the amount of data to be treated within a gives subdomain;

- Reduce: the reduce operation, can be done in multiple stages and depends on the operation. Operations like MAX, MIN, PRODUCT and SUM can be done locally before transmitted. The same for AVERAGE, MIN_LOC and MAX_LOC, but these operations include further information, like the number of data elements included, the rank where is the minimum and rank where is the maximum, respectively. User defined operations can also be used, as the reduction is done at the native side by means of a C function pointer. However, they cannot depend on extra information.

- All-to-All: instead of centralizing the data at the coordinator, the data is centralizes in the most external *clustering* component involved and dispatched to all process as a broadcast message. This reduces the number of copies and indirections on message transmission.

6.7 Chapter Conclusion

In this chapter, we have presented a prototype implementation for the proposal presented in the section 4. This prototype is based on a GCM-based infrastructure and takes profit of the ProActive deployment with some introduced extensions to deploy MPI applications and the runtime. Some principles have guided the development of this prototype, and were important in two aspects: build an user-friendly environment, that provides good performance on the execution of point-to-point and collective communications among processes on the same cluster and externally.

The approach adopted make possible to deploy a gridified MPI application with a minimum knowledge about resources. The default configuration guarantees a reasonable performance, but optimizations can be configured to improve performance. Performance was always the focus on the implementation of this prototype, and for this reason, it includes some important principles, like enforcing the usage of MPI

communication for inner-cluster communications, reduce indirections at the component level and reduce the time needed to perform point-to-point and collective communications.

The next chapter evaluates this prototype through some microbenchmarks and experiments.

7 EVALUATION

This chapter presents the evaluation of the proposed model and prototype. Initially, we present some simple benchmarks to measure the overhead introduced by the use of the component support in synchronous and asynchronous point-to-point and different collective communications. After, we present some experiments based on some applications developed/executed using our framework, comparing them with pure MPI versions. This chapter ends with a qualitative comparison between the developed framework and related tools.

7.1 Experimental Environment

The benchmarks and experiments were conducted in the Grid5000 testbed, on 8 different clusters spread around 4 different sites in France (in Sophia Antipolis, Rennes, Lyon, and Grenoble), composed by machines of different processor architectures (Intel Xeon EM64T 3GHz and IA32 2.4GHZ, AMD Opterons 2218, 246 and 248) and memory from 2GB to 4GB by node. Internally, 2 of the clusters have Myrinet-10G and one of them Myrinet-2000. The interconnection among the different clusters is done through a dedicated backbone with 2,5Gbit/s links

The native executions make use of the MPICH 1.2.7p library and the ProActiveMPI ones make use of MPICH v1.2.5, ProActive v3.9 and Java Sun SDK 1.6.0_01. When we mention cross-cluster communication, it means that the communication takes place in one site and multiple clusters while cross-site means that the underlying MPI environments were running in multiple clusters, geographically separated. Also, when running in more than one cluster and/or site, the number of nodes were divided equally among the clusters, for instance: in the experiments with 60 nodes, the MPI-local configuration consists in 60 nodes placed within a single cluster, the ProActiveMPI-2clusters configuration has 30 nodes on each cluster and in the ProActiveMPI-2sites-2clusters 15 nodes on each of the clusters. Remaining nodes are always deployed on the last cluster

The numerical results are calculated through a simple the average of multiple executions: a hundred for short executions (a couple of seconds) and ten executions on long experiments (those that take around couple of minutes or more), for each configuration and scenario. In the case of the benchmarks executed in multi-cluster experiments, we also include the standard deviation, due to the fact that the network that connects the sites of the grid is a shared resource. The speedup is calculated by the formula $speedup = Time_{sequential}/Time_{parallel}$ and is useful to verify the efficiency obtained out of a parallel application.

7.2 Microbenchmarks

This section presents the benchmarks of the main communication operations in different scenarios as well as an evaluation of the overhead introduced by the component layer. In order to have a more realistic view of the performance, we compare execution times with the same kind of benchmarks with MPI. But it is important to remember that the goal of the work is not to replace existing MPI implementations.

Initially, we present an evaluation of performance of synchronous and asynchronous point-to-point communication with and without the proposed optimizations (section 6.6). Then, we propose a simple classification of the collective communication and some benchmarks with the different classes of collective operations.

7.2.1 Point-to-Point Communication

In the first microbenchmark, we evaluate the point-to-point communication in different scenarios. But, first, we evaluate the overhead measured in single-site execution for synchronous and asynchronous operations.

The figure 7.1 shows that despite of the existence of a control to decide how to handle messages, the overhead on communication that takes place within a single cluster is negligible. The figure 7.2 shows that the same behavior can be expected on asynchronous communications that are, in general, faster than blocking ones. Even if these results were foreseeable, it is important to emphasize that the runtime does not impact negatively in performance of communication within a single cluster because this kind of communication is very recurring.

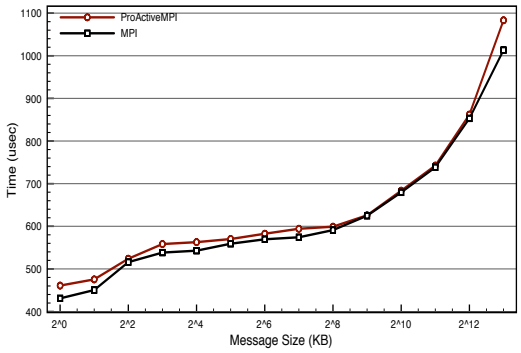

Figure 7.1: MPI_Send - MPI_Recv: single cluster performance

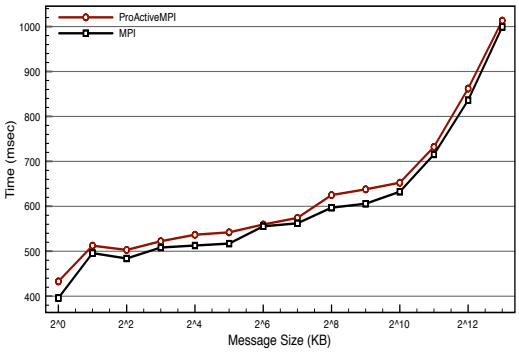

Figure 7.2: MPI_ISend - MPI_IRecv: single cluster performance

The figure 7.3 shows a performance comparison for the point-to-point communication between two clusters (located in Sophia-Antipolis) and the figure 7.4, the performance of the non-blocking point-to-point communication. These graphics compare the point-to-point communication in three different contexts: (i) pure MPI, which is possible due to the configuration of the Grid5000 that enables all-to-all nodes access; (ii) component-based communication, having messages forwarded through components, that is the behavior if direct communication is not possible and (iii) component-based version optimized with the tensioning technique. The figures figure 7.5 and 7.6 show the performance of the point-to-point communication between two sites (Sophia Antipolis and Grenoble).

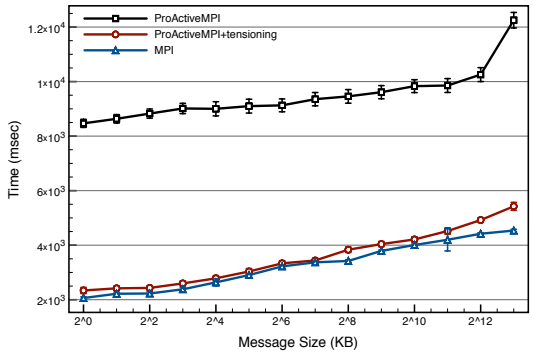

Figure 7.3: MPI_Send - MPI_Recv: cross-cluster performance

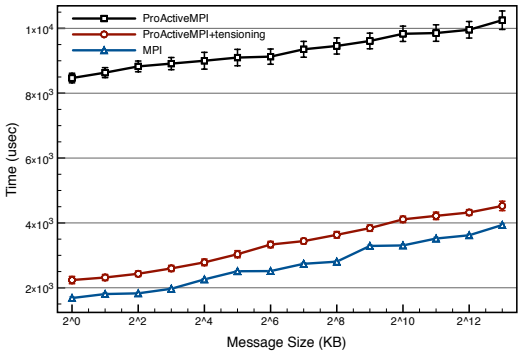

Figure 7.4: MPI_ISend - MPI_IRecv: cross-cluster performance

In all the four benchmarks, the difference in performance between the versions that perform direct communication and the version that uses the component infrastructure to forward messages is remarkable. This difference comes from the obvious fact that the communication through components requires the messages to be sent to the JVM stack, and communicated through network, then from the receiver JVM to the native process. Even if direct memory access (given by the java.nio package) enables a zero-copy transfer between C and Java, the messages must be copied from component to component in the hierarchy.

The difference in performance between ProActveMPI versions justifies the principle 2 (section 6.1) of the usage of direct communication whenever possible. It also emphasizes the usefulness of *tensioning* optimization technique in component-based software architectures. Even if ProActiveMPI communication is slightly slower than pure MPI communication, the usage of *tensioning* have enabled to perform communication in a very flexible way, without all the cost of indirections.

The main reasons for the differences in time between cross-cluster and cross-site communication is the bigger latency and smaller bandwidth in the network between sites than the link between two clusters.

The vertical error bars represent the average difference of the samplings, in relation to the global average. The bigger this bar is, the bigger was the difference in communication time from one execution to another. In the case of Grid5000, this happens specially in cross-cluster communications because they imply on the usage of a shared backbone. Some bigger error bars are mainly due to transient behaviors on network performance and happened in less than 3% of the cases.

The peak around 4MB for the point-to-point communication through the component is due to the RMI communication mechanism, as already verified in (JAGANNADHAM; RAMACHANDRAN; KUMAR, 2007).

7.2.2 Collective Communication

As already discussed in the section A.11.2, the collective operations follow a completely different mechanism. For this reason, we evaluate the collective communication separately. This section present some benchmarks over the execution of collective calls on a multi-cluster base grid environment.

In order to avoid redundancy on the benchmarks and discussion of results, we have classified the collective communication primitives in three different groups,

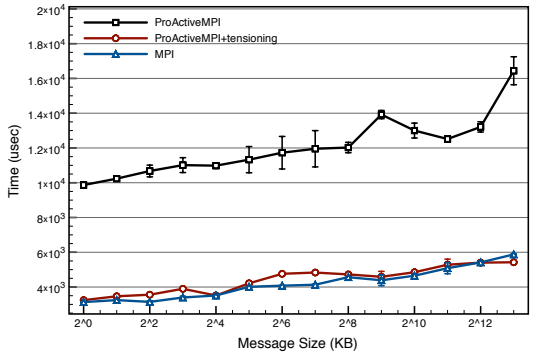

Figure 7.5: MPI_Send - MPI_Recv: cross-site performance

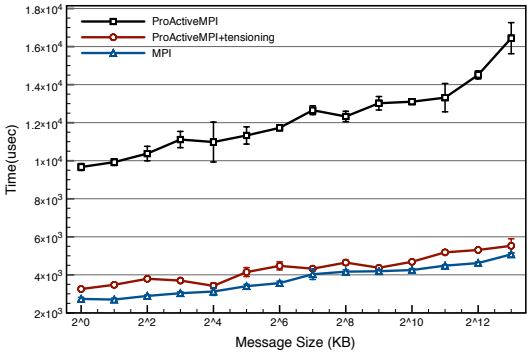

Figure 7.6: MPI_ISend - MPI_IRecv: cross-site performance

according to the similarity of collective communication mechanism:

1. Broadcast-based: this group include all the collective communications which have a one-to-many communication. Besides of the Broadcast mechanism, this groups includes all the Scatter operations. The results regarding this group will be verified by means of a Broadcast operation.

2. Gather-based: this group includes all the collective communications which have a many-to-one communication. Besides of the Gathercast mechanism, this groups includes Reduce operations. The results regarding this group will be verified by means of a Gathercast operation.

3. Gather-based plus Broadcast-based: this is the case of more complex operations, which imply on a first step that gathers data and a second step, that dispatch the results, such as Barriers, All-to-All and AllReduce communication. The results regarding these collective communications will be verified by means of an AllReduce operation.

The figures 7.7, 7.8 and 7.9 show the performance of collective operations is ProActiveMPI and MPI considering different resources configuration: single cluster, 2 and 3 clusters located in different sites. As in the point-to-point communications, the execution of the three communications primitives on a single cluster, takes about the same time in both cases(MPI and ProActiveMPI) because in fact they both rely on the same MPI communication mechanism.

In general, the pure MPI communication involving multiple environments is slower than ProActiveMPI communication. This happens because the collective operations in MPICH follow a tree-based algorithm when there are more than 4 process involved. The execution of these operations without taking into account resources topology usually lead to performance issues because, at some point of the operation, unnecessary and expensive inter-cluster communication are done.

Differently, the ProActiveMPI takes into account resources topology, reducing unnecessary inter-cluster communication. For this reason, it performed better in multi-cluster grid environments. But, the behavior changes depending on the operation:

- In the case of the broadcast operation (figure 7.7), the execution in 3 clusters was faster than in 2 clusters because it increased the amount of operations happening at the same time;

- On the other side, the gathercast operation (figure 7.2.2) performed better with 2 clusters because it reduced the need of an extra copy for the execution of a gather at cluster level, before the propagation;

- In the case of AllReduce (figure 7.9), the ProActiveMPI version performed even better that the executions in single environment. This occurred because the reduction operation happened in parallel in each of the clusters before being propagated to the root of the operation, which also reduced bottlenecks.

As in point-to-point communication, the error bars are bigger when more clusters/sites are involved. This is a natural evidence that communication involving shared network are more susceptible to transient affectations.

All in all, the collective communication curves have a similar shape. This comes from the fact that, at the cluster level, the runtime performs a plain MPI communication that, in general, is the most expensive operation because it involves a bigger number of process and a centralized entity.

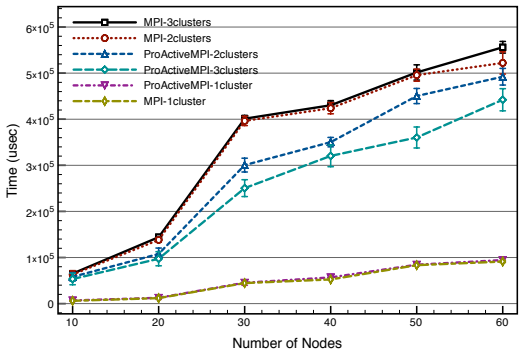

Figure 7.7: Broadcast: cross-cluster performance

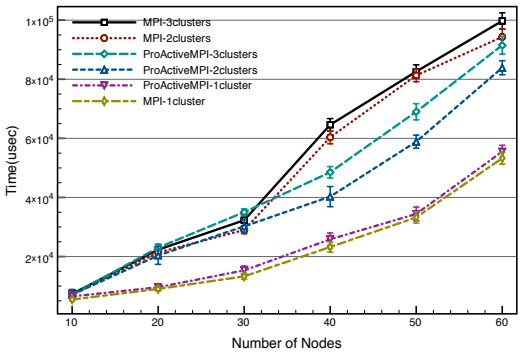

Figure 7.8: Gathercast: cross-cluster performance

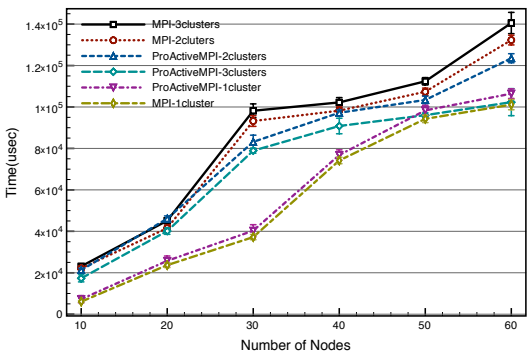

Figure 7.9: AllReduce: cross-cluster performance

7.3 Experiments

In order to have a more practical and complete evaluation, we developed some application, which represent different classes of applications: (1) a monte-carlo simulation to compute Π (3.14159265...), representing the embarrassingly parallel master-worker applications; (2) a hierarchical parallel mergesort, representing the Divide-and-Conquer approach to build parallel applications; and (3) the Poisson3D solver representing non-embarrassingly application based in domain decomposition.

The idea of choosing these applications is that we could expect a similar behavior in applications that follow the same approach. The following section presents, in more details, each of the applications and how they were developed along with some code examples. After, we analyze the obtained results.

7.3.1 Applications

7.3.1.1 (1) Monte-Carlo Simulation

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used to simulate physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm or equation. As the precision of the computation is based in the number of iterations, Monte Carlo is a method suited to calculation using parallel resources like clusters or grids.

In our case, the Monte Carlo method is used in a simple synthetic application that calculates Π (3.14159265...). It consists on the generation of pseudo-random numbers between 0 and 1 representing a given point on the figure 7.10 and the computation of Π is based in the verification whether the points are inside or outside of the circle.

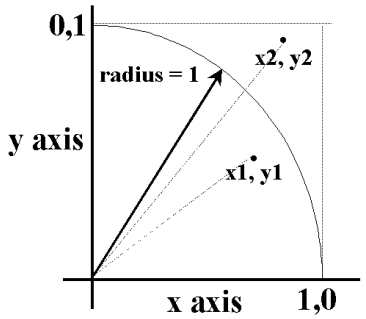

Figure 7.10: Pi Computation Scheme

The area of a quarter of circle is done by:

$$A = \frac{1}{4} * \Pi * r^2 \tag{7.1}$$

Considering the formula of the distance between 2 points x and y as:

$$D = (x_2 - x_1)^2 + (y_2 - y_1)^2. \tag{7.2}$$

Considering that the $A = D^2 = 1$ and P_{in} the number of points within the given area and P_{out}, the points outside, we can statistically compute Π as being:

$$\Pi = \frac{r^2 * \frac{P_{in}}{P_{out}}}{\frac{1}{4} * r^2} = 4 * \frac{P_{in}}{P_{out}} \tag{7.3}$$

The implementation of the parallel application follows a master-worker pattern, on which a master process assigns independent set of tasks to each worker processors and retrieve partial result, computing the final result as an average of the partial ones.

This application is a typical CPU-bound application and communication does not play an important hole on the elapsed time. Nonetheless, in order to have a fairer comparison among the different applications, we have developed two different algorithms:

- (i) flat version: this version has a simple master-worker model, on which the tasks are calculated by the workers and retrieved through a reduce operation. For comparison effect, we also evaluate a flat pure MPI version. The appendix C presents a snippet of the source code that shows the communication scheme.

- (ii) hierarchical version: this version intends to reduce inter-site communication by introducing one-level hierarchy. The results are first gathered in the first worker of each site before they are sent to the master. The appendix D presents a snippet of the source code that shows the communication scheme.

7.3.1.2 (2) Hierarchical Mergesort

Mergesort is a well-known sorting algorithm developed by John von Neumann (BRON, 1972) and have average and worst case performance of $\Theta(n \log n)$. Mergesort sorts by employing a divide-and-conquer strategy to divide a list into two sublists. The steps are:

1. Divide the unsorted list into two or more sub-lists of about the same size.

2. Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned.

3. Merge the two sorted sublists back into one sorted list.

The figure 7.11 summarizes these steps for a small array.

Due to its divide-and-conquer nature, mergesort can be easily parallelized. Two different versions of the mergesort were developed:

- (i) flat version: even if merge-sort has a dived-and-conquer organization, this version follows a master-worker pattern (1 level of tasks) on which the array of numbers to be ordered is scattered among all the workers in sub-arrays of equal size. After, the master process waits for all ordered sub-arrays and merge them as they arrive, by using an insertion sort algorithm. This implementation does not follow strictly the classical algorithm as the granularity is given by the number of nodes, rather than dividing data until two elements are to be sorted

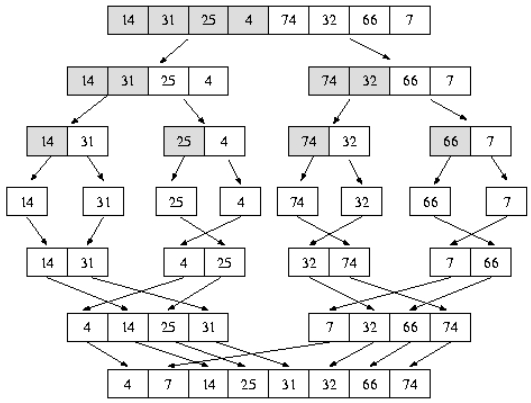

Figure 7.11: Mergesort Example

(trivial sort). The figure 7.12 shows the communication pattern, adapted to the flat version.

The appendix E presents a skeleton of the application, containing the main communication primitives. On this applications, we have evaluated the possibility of running unmodified applications

- (ii) hierarchical version: this version implements the mergesort by using a hierarchical approach (with 3 level of tasks). Initially, the array of numbers is divided in n sub-arrays, being n the numbers of sites. After, this sub-arrays are divided again within the site for each cluster and then within the cluster to the nodes (3 levels of tasks). The main benefit of the hierarchical version is that the merge process happens in parallel in each of the clusters. The figure 7.13 depicts the hierarchical algorithm.

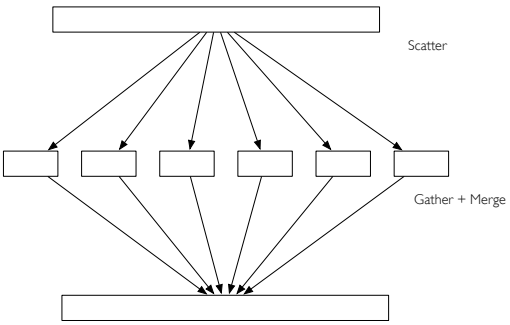

Figure 7.12: Flat Mergesort Scheme

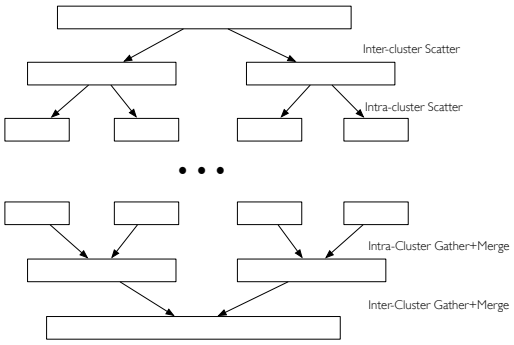

Figure 7.13: Hierarchical Mergesort Scheme

The appendix F presents a snippet of the source code that shows the communication scheme. This code may seem clumsy, but intermediary nodes execute almost the same code and this could be generalized.

7.3.1.3 (3) The Poisson3D Solver

Poisson's equation is a partial differential equation with broad utility in electrostatics, mechanical engineering and theoretical physics. Different from previous applications, the Poisson3D algorithm does not have a master-worker organization.

This means that there is not a process that coordinates the computation by dispatching tasks to the others. Instead, each process handles a piece of the entire matrix and communicate with process that handle neighbor pieces of the matrix.

Our Poisson3D implementation uses an iterative Jacobi method. In general lines, the idea is to partition an entire mesh (the domain) into several sub-domains, and a global solution is recovered by a succession of solutions of the independent sub-domains. It consists on a bulk-synchronous application given by the following steps:

1. Concurrent computation: each process executes the Jacobi method over its own sub-domain;

2. Reduction of results: each process calculates an error and all the processes reduce their errors in order to determine the smallest error. A threshold determines the stop criteria (in order to force a deterministic behavior for the performance evaluation, the stop criteria is given by a fixed number of iterations);

3. Update of sub-domain borders: each process updates its borders with neighbor processes and proceed to the step 1.

Considering the steps enumerated above, we can clearly identify the main communications involved: in the step 2, we have a global Reduce operation, and in the step 2 an 'Update' operation, which consists on a Scatter/Gather among neighbor processes.

As seen in the section 4.1, intra-cluster communications are faster than inter-cluster ones, as they take advantage of local high-speed network. For improving the overall application performance, we investigated different partition and mapping schemes according to resources organization (provided by extended API). The figure 7.14 shows two examples of partition schemes.

After experiments, the best option has proved to be the partition of the mesh in slices instead of a cubic partition (see Figure 7.14). Thus, with a correct sub-matrix mapping to resources, we ensure at most two processes per cluster will need to perform such communication.

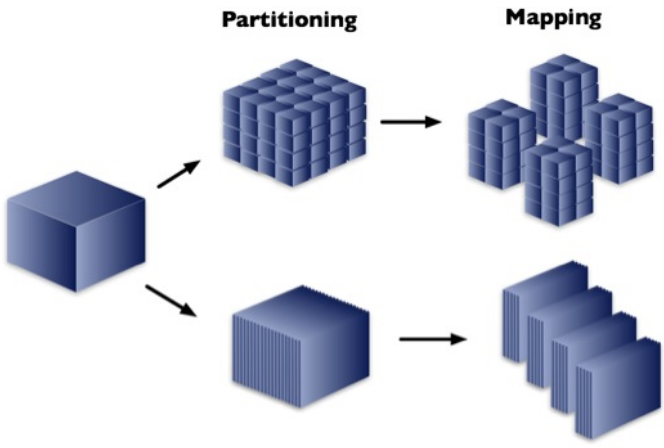

Figure 7.14: Poisson 3D Mesh Partition/Mapping Approaches

Two versions using this algorithm are compared on this experiment: the original pure MPI version and a *gridified* version that make use of the defined API to properly partition the mesh and organize communication hierarchically.

These two versions differ essentially in two points:

- the partition of the ProActiveMPI versions takes into account resources distribution minimizing inter-cluster communication, while the pure MPI one does not take into account resources position because topology information is not provided by the runtime;
- the communication layer used in cross-cluster communication is different: the communication is handled by the runtime while pure MPI is based entirely in MPI communication (only possible because the Grid5000 allows it);

7.3.2 Experiments Results

This section presents the results and analyze of the experiments described in the section 7.3.1. The performance obtained in most of the experiments depend on a large number of variables, from resources location, instantaneous network load, data partition, modifications/adaptations in algorithms and obviously the underlying framework. Differently from the section 7.2, the discussion of the results provided here are based on author's experiences and intuition but could be eventually interpreted differently.

7.3.2.1 Monte Carlo Results

The Figure 7.15 shows the execution time for the Monte Carlo Simulation executing 10^8 iterations in two different implementations and different execution scenarios: a pure MPI version of the application on a single cluster and an application that uses the proposed extensions on 2 and 4 clusters. For the sake of simplicity, the results regarding MPI cross-cluster execution and flat organization were suppressed as they were very similar to flat executions.

On the figure 7.15, we can notice that all the curves have almost the same shape and they are pretty close to each other, which means that there is no significant overhead induced by the component framework during the computation. Later we will see that this behavior is highly dependent of the application and the design of the the communication algorithm as well as the amount of communication done.

The difference in execution time between the executions in 2 or 4 clusters are probably due to larger latencies in the communication with the third and fourth cluster, that were geographically distant from the other 2. Also, we can notice that there is one specific configuration (10 nodes in 3 clusters) where the execution with 3 clusters had a performance degradation. This happened because one of the clusters had 4 processing nodes while the other 2 clusters had just 3. As the processing load was equally divided through the clusters, the power equivalent to one of the nodes was wasted, as the execution time is that of the slower cluster.

From the figure 7.16, we can notice that, for this experiment, our framework can be considered as scalable as the MPI implementation used (MPICH-1.2.7b). Some empirical experiments have shown that, for the monte carlo simulation, this behavior stills valid even for hundreds nodes and that the degradation in performance in such cases was comparable to that of the pure MPI version of the application. The main aspect that helped to keep a good overall performance, even for the flat version, is

that the monte carlo application is essentially CPU-bound and does not depends on the communication of a large amount of data.

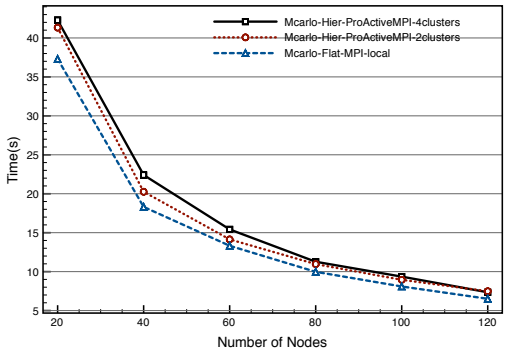

Figure 7.15: Monte Carlo Performance

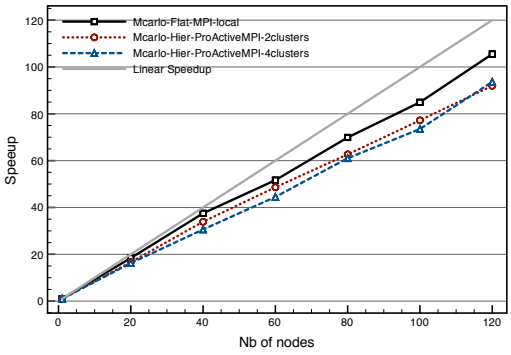

Figure 7.16: Monte Carlo Speedup

7.3.2.2 MergeSort Results

The figures 7.17, 7.18 and 7.19 show the execution time for the MergeSort application, ordering the same 10^7 random long numbers (around 38MB of data) in different contexts: pure MPI execution on a single cluster (flat algorithm) and execution through multiple sites/clusters of the flat and hierarchical version. In all the executions, the number of computing nodes was equally distributed on the clusters. When the number of clusters is mentioned, it means the number of clusters present on each site and not the global number of clusters. The execution times does not take into account the time needed to read data from the input file. The results for multi-cluster executions start from 4 nodes to ensure at least one node on each cluster.

Unlike the monte carlo application, the experiments conducted with the mergesort application showed a bigger degradation in performance when using multiple clusters. However, we must highlight that we are enabling the coupling of more than one cluster in a single MPI computation. This means that we can aim at gathering a bigger computational power out of multiple clusters for a single execution. Nonetheless, the executions in a single cluster is presented together with the other executions as a parameter.

When comparing the figures 7.17 and 7.18 we can notice the importance of taking into account the topology on design of the parallel algorithm. In the flat version, there are some configurations where the ProActiveMPI version takes about 8 times more that the pure MPI version, against 2.5 times in the worst case of the hierarchical version. This happens because, in the flat version, a large amount of the communication process happens in a WAN context (so with a greater latency and smaller bandwidth) and we need 'n' big messages to spread the data to 'n' nodes computing and 'n' messages to gather the ordered arrays back, while in the hierarchical version, we just need a number of messages equivalent to the number of sites. Even if the messages are bigger in the hierarchical version, the impact in performance stills smaller.

Another point that impacts positively on the performance of the hierarchical version is that the merge process happens in parallel on each cluster, and after on each site, while in the flat version, this is a centralized process. According to the

Amdhal's law, the execution time of an application is given by the sequential part plus the parallel part and so, the bigger the sequential part is, the less an application can scale.

The impact on the parallelization of the merge process can be seen more clearly on he figure 7.19 that compare different executions with the same resources organization (2 sites, with 2 clusters on each site and nodes from 1 to 16 on each cluster). From this comparison, we can see that the flat version just have the overall computing time reduced until 32 nodes, and in the hierarchical version, we could have an improvement on the performance up to 128 nodes and probably even more. For a small number of nodes, the flat version probably performed better due to the fact that, in the hierarchical version, the inclusion of multiple levels imply on more time spent to transmit the data through the levels in the hierarchy.

Yet in the figure 7.19, we can see that the mergesort algorithm does not scale well in general. This happens because of two main reasons: the communication time is significant on the overall time and the merge operation is inherently a sequential operation, that becomes expensive as the number of workers increases, even if it is softened through the hierarchical version.

Even if we could not keep the good speedup of monte-carlo (figure 7.16) for the mergesort, it does not means that the developed prototype does not scale well. A proof of that is that even the pure MPI, that is considered highly scalable, did not obtained a good speedup. Some other experimental executions have shown that the speedup can be a little better when ordering a big amount of numbers, because the computation time encompasses the communication costs. Nonetheless, the merge process prevents the obtention of a good performance as it is proportional to the amount of numbers to be ordered.

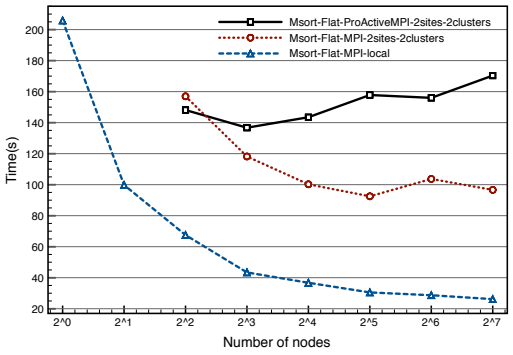

Figure 7.17: Flat Msort (i) Times

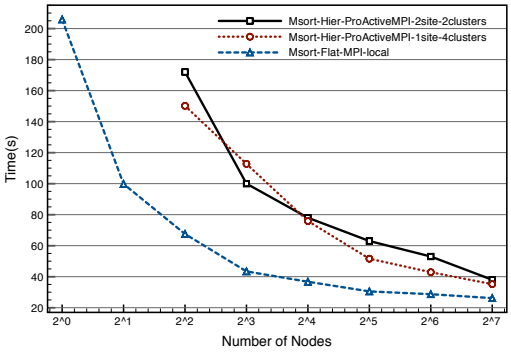

Figure 7.18: Hierarchical Msort (ii) Times

7.3.2.3 Poisson3D Results

The figure 7.20 shows the execution time for the P3D application, computing a regular mesh containing 1024^3 elements (around 8GB of data) over 100 iterations on the 8 sites described in the section 7.1, with 2 clusters on each site and nodes from 2 to 32 on each cluster. The results start from 16 nodes (2 by cluster) to avoid usage of OS memory swap on computing nodes: some nodes present just 2GB of memory and 4 cores, so the allocation of 512MB per computing node, requires at least 16 nodes for the execution of the application.

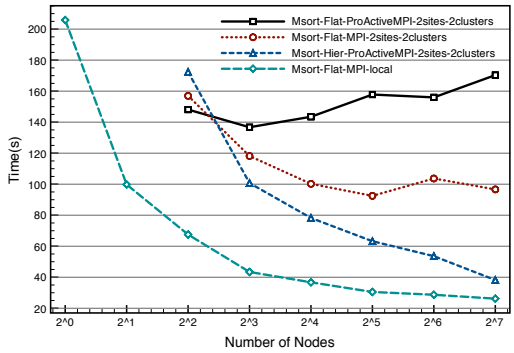

Figure 7.19: Comparison between flat and hierarchical ProActiveMPI Msort versions

The results in the figures 7.20 show that, even for a small number of nodes, the application can profit from the hierarchical approach. This is possible first, by reducing the data shared and communicated among clusters, but also by the parallel execution of operations like the global reduce, that is very time consuming (more about AllReduce performance evaluation in the section A.11.2, figure 7.9).

Comparing the two executions of the flat version (MPI and ProActiveMPI), the MPI version performed better because it runs without any overhead of the component infrastructure, completely in native mode. However, we must call attention to the fact that most of the grid infrastructures do not allow unrestricted inter-cluster communication, as it is was possible in the Grid5000.

This execution takes into account a sliced partition, but different partitions schemes (e.g. cubic partition) lead to important changes in execution times.

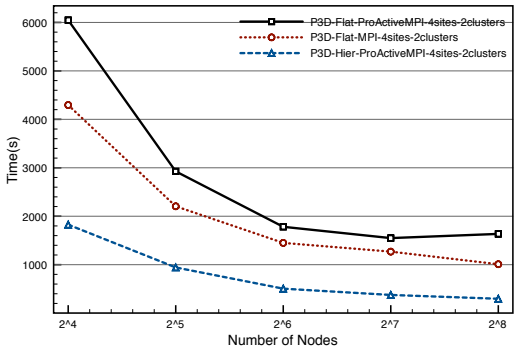

Figure 7.20: Comparison between flat MPI, flat ProActiveMPI and hierarchical ProActiveMPI versions of the Poisson3D application

7.4 Comparison with existing tools

The main intention of this work is to offer support to hierarchical communication in grid environments by means of extensions to the MPI standard and a component-based support. As such, our "tool" also benefits from a number of other features required to run applications in grids.

So, besides of the quantitative performance evaluation, it is relevant to do a qualitative comparison of the work presented in relation with existing related tools

(Chapter 3).

The table 7.4 summarizes this comparison and the criteria is listed as follows:

1. Support to network and grid protocols/middlewares: protocols and/or middlewares may differ from one grid platform to another. This criteria also have relation with requirements presented by some tools, like MPICH-G2 and H_2O, as some of them relies on top of grid middlewares;

2. resources allocation, access and management: by definition, grids are composed by remote shared resources. The usage of these platforms usually depends on the allocation, access and management of resources. Some of the tools present these features as a built-in functionality, while others relies on third parties middleware.

3. data management: in general, high performance applications have need of input data and/or generate output data. At least, the binaries are necessary to start the computation. Because of this need, some of the tools provide mechanisms to ease data management. Others, just let users handle this kind of issues.

4. automatic forwarding/tunneling of messages: by definition, grids can be composed of resources located in different domains, sometimes behind firewalls or even having private network addresses. While some tools assume all-to-all accessibility, some others offer the necessary support to automatically forward messages in the case where the all-to-all access in not possible. Some of them also offer the possibility of tunneling of messages, so enabling MPI applications to run in firewalled environments.

5. topology-aware communication: in this work we identified topology-aware communication as a requirement to obtain high performance in heterogeneous distributed environments, by avoiding slower network connections. For this programming pattern to be possible, some of these tools offer topology information to help users to program applications in a more topology-aware way, while some others support topology-aware communication, but by hiding it from the users in the middleware.

6. API: there are two main approaches to enable MPI to run in grids: either the tools intend to support unmodified MPI applications by means of the underlying support that hides from the user grid related issues or they offer an API to deal explicitly with grid characteristics.

From the comparison table 7.4, we can see that besides of offering abstractions and primitives to develop hierarchical grid-aware application, our solution copes with grid requirements and offer most of the features presented in other tools.

7.5 Chapter Conclusion

In this chapter, we have presented the evaluation of the current work from different points of view. This evaluation started with an evaluation of the basic communication mechanism, then some application have shown some empirical results and finally a qualitative comparison in relation to related works.

Features \ Tools	PACX-MPI	MPICH-G2	MPICH-Mad	H20	ProActiveMPI
1. Support to network/ grid protocols/ middleware	network protocols	Globus only	network protocols	messaging protocols	network protocols (through MPI) and grid middlewares
2. Resources Management	no	yes (through Globus GRAM/DUROC)	no	no	yes (PBS, LSF, OAR, Globus, gLite, ...)
3. Data Management	no	yes (through Globus sandboxing and GlobusIO)	no	no	ProActive File-Transfer
4. Forwarding/ Tunneling of Messages	Forwarding/Tunneling	no	Forwarding/Tunneling (configured by hand)	Forwarding/Tunneling	Forwarding/Tunneling (configured by hand)
5. Topology-aware Communication	no	yes (implicit)	no	no	yes (explicit and implicit)
6. API	no	yes (new MPI attributes)	no	no	yes (new MPI Comms and primitives)

Table 7.1: Comparison of the developed prototype with related tools

The microbenchmarks result have shown that the overhead generated by the component layer is negligible when considered just one cluster. However, in multi-cluster infrastructures, they impact in the overall performance of applications. This fact, this highlighted the importance of optimizations in runtime to enable direct communication they are possible. The collective communication support has also proved to cope with grid issues, yet offering a reasonable performance.

The results obtained with the experiments have shown that non-embarrassingly applications must be *gridified* in order to really take profit of resources hierarchy and, as a result, these applications can scale more easily on grid resources.

The qualitative analysis showed that, even with a completely different approach, our solution came up with most of the features offered by related tools. This reveals the general character of the solution, which could be easily used in different contexts, providing solution to the execution upon different resources specifications and applications needs.

8 CONCLUSIONS AND PERSPECTIVE

The inherent distributed, heterogeneous and hierarchical nature of multi-cluster grid environments shifts the emphasis in many programming issues, namely the need of an adequate programming model. Because of the wide acceptance of message passing and MPI as the standard paradigm to develop high performance applications, the idea of using MPI in grids has always been subject of investigation nowadays.

Up to now, the main approach to deal with MPI and grid computing has been that of executing unmodified MPI applications. On the other side, parallel algorithms must be adapted to reflect grid topology in order to obtain a good performance. Despite of this need, we identified a lack of mechanisms and abstractions to design and develop grid-aware hierarchical MPI applications. Also, we believe that the design of MPI is very static to cope with grid characteristics like dynamism and heterogeneity of resources. Differently, the component-based paradigm offers the adequate support to address these grid requirements and also the possibility of encapsulating native legacy codes.

In this context, we proposed a hybrid model through extensions in the MPI standard to address hierarchical message-passing and a grid component-based framework supporting the newly introduced features. The developed prototype takes profit of the ProActive grid middleware that already offers support to grid deployment, native code wrapping and a reference implementation of the GCM.

From the results obtained in the evaluation of this prototype, we conclude that the overhead introduced by the components is not negligible, but inside of the expected. However, we can expect the benefits to grid applications to bypass the overhead generated. But, in order to profit of resources hierarchy, applications must be (re)designed in a hierarchical fashion. This is not a simple task by any means, and the applications developed to evaluate this work have showed this. Depending on the application approach, a hierarchical work-stealing could be preferred, as suggested in (PEZZI et al., 2007).

The research that is being developed as the continuation of the present work (more details in next section) points to a new direction on the development of parallel SPMD applications and this seems to provide a good solution to the some issues present on this work, namely the introduction of dynamicity at runtime level and abstractions to ease development of hierarchical MPI applications.

8.1 Research Perspectives

The main outcome of the presented work was the introduction of high-level communicators and a component-based runtime supporting the introduced mechanisms.

However, we have seen that the design and development of real size applications is a very complex task. The same need was detected by some other research projects focused in applied mathematics, more specifically on the design of numerical methods for the computer simulation of complex physical phenomena related to two application domains: computational electromagnetism (CEM) and computational fluid dynamics (CFD).

As a result, the DiscoGrid Project was started in 2007, funded by the French National Research Agency (ANR) for 3 years. The DiscoGrid project aims at studying and promoting a new paradigm for programming non-embarrassingly parallel scientific computing applications on distributed, heterogeneous, computing platforms. The target applications require the numerical resolution of systems of partial differential equations (PDEs) modeling electromagnetic wave propagation and fluid flow problems. More importantly, the underlying numerical methods share the use of unstructured meshes and are based on well known finite element and finite volume formulations.

The main objectives of the DiscoGrid project include:

- the definition of a high-level API to develop domain-decomposition based simulations that abstract completely the complexity of MPI;
- a runtime capable of suffering adaptations coping with the evolution of a changing environment, offering inter-cluster communication with high performance;
- a mesh partitioning tool capable of taking into consideration resources topology to reduce data shared among processes in different clusters/sites;
- the development of real-size simulation software based on the introduced paradigm.

The present work already offers the solution to some of these needs and was investigated as a first prototype having served as the base to develop a simple application (the Poisson3D). But the abstraction of high-level communicators was not considered enough to support an easy development of scientific domain-decomposition applications. Besides, a full solution requires a more advanced component model, capable of adapting to different resources organizations and optimizations in execution-time.

On this new component model, we are currently exploring the concept of separation of concerns of component models to define the SPMD programming approach as a composition activity, rather than strong code coupling within the application logic. The idea is to see parallelism and group communication mechanism as non-functional aspects of the applications. By doing so, users take profit of the high-level abstractions proposed by the DiscoGrid project and let the runtime handle the entire communication and execution process, that can be entirely defined and optimized in execution-time through non-functional aspects of components.

Even if the goals of the DiscoGrid project implied on the definition of a different user interface and the evolution of the presented runtime, some of the concepts proposed by the current work were adopted by the DiscoGrid project:

- the usage of components as the support to develop the runtime: different partners have focused on other component models because the only GCM implementation is based in Java and the communication implemented entirely

in RMI, which can not be easily integrated with tools developed in other languages;

- the collective communication mechanisms, implemented at component level;
- the concept of hierarchical identifiers: the DiscoGrid project decided to not keep the idea of high-level communicators, because the basic concept of communicators does not cope with dynamic environments, but processes have the same identification mechanism proposed in this work;
- the deployment mechanism: the extensions to the ProActive deployment are being currently used to deploy DiscoGrid applications;

For this reason, we believe that this work brought important contributions and that it can be seen as a successful first illustration of a new parallel programming paradigm: the hierarchical message passing. Besides, the prototype can be seen as a seed for the development of SPMD component-based software.

9 ACKNOWLEDGEMENTS

We thank CAPES and INRIA DREI (Direction des Relations Internationales) for the student financial support. This work was also partly carried out under the DiscoGrid project funded by the French ANR (Contract ANR-05-CIGC-005) under the framework of the program Calcul Intensif et Grilles de Calcul.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr)

APPENDIX A RESUMO DA DISSERTAÇÃO EM PORTUGUÊS

Nos últimos anos, a computação em Grades emergiu como uma forma de agregar recursos distribuídosde múltiplos domínios administrativos. Devido a natureza heterogênea e distribuída dos recursos, a computação em Grade aumentou a importância dada a requisitos específicos da computação distribuída (como descrito na seção 2.1.3).

A.1 Problemática

Vários modelos de programação já foram propostos para programação de Grades. Apesar disso, até agora, nenhum deles responde a todos os requisitos, principalmente dinamicidade, escalabilidade e desempenho. Como mencionado em (FOSTER; KESSELMAN, 1999),

> *Ambientes de Grade exigem uma nova percepção dos modelos de programação existentes e, provavelmente novos modelos que correspondam às características de aplicações e ambientes de Grade.*

De uma forma diferente, na área de computação de alto desempenho em características, o modelo de passagem de mensagens se tornou um verdadeiro padrão com um grande número de bibliotecas e aplicações legadas. Por esse motivo, o uso de MPI para o desenvolvimento de aplicações para Grades tem sido investigado em projetos de pesquisa e no meio industrial.

Por não ser um modelo de programação de alto nível, o modelo de passagem de mensagens carece de mecanismos que permitam o desenvolvimeno de aplicações para Grades. Além disso, o padrão MPI foi concebido para utilização em ambientes de cluster, não possuindo primitivas adaptadas para programar ambientes de Grade envolvendo múltiplos domínios administrativos, que são inerentemente hierárquicos (PEZZI et al., 2007). Opostamente ao modelo de passagem de mensagens, modelos baseados em componentes são capazes de oferecer funcionalidades de outros modelos de programação (MOREL, 2006) (p.ex. modelo de passagem de mensagens, objetos distribuídos, serviços, workflow, etc.), além de contar com a capacidade de encapsulamento de código. Então, pode-se argumentar que o modelo de componentes é mais adequado à programação de middlewares para Grades computacionais.

A.2 Objetivos e Contribuições

Os principais objetivos desse trabalho de mestrado são:

- obter o alto desempenho, associado à alta aceitabilidade do padrão MPI, melhorado com extensões intuitivas que permitam à desenvolvedores projetar e desenvolver aplicações para grades ou "gridificar" aplicações existentes;

- com a flexibilidade de um runtime baseado em componentes, modelando a hierarquia dos recursos disponíveis e oferecendo suporte à comunicação intercluster.

As principais contribuições desse trabalho incluem:

1. a definição de extensões intuitivas ao padrão MPI para comunicação hierárquica;

2. um suporte simplificado ao lançamento e controle da execução de aplicações MPI em múltiplos clusters;

3. desenvolvimento de um framework baseado em componentes que ofereça suporte ao modelo de passagem de mensagens em ambientes multi-cluster através de uma camada de software que modele a organização dos recursos;

4. desenvolvimento de aplicações que sirvam de teste e comprovem a utilidade das extensões introduzidas nesse trabalho.

O principal objetivo desse trabalho não é competir com implementações existentes do padrão MPI e tampouco substituir o padrão MPI. O objetivo é oferecer uma alternativa mais adaptada ao desenvolvimento de aplicações para grades e de oferecer um runtime que suporte essas novas funcionalidades.

A.3 Projetos de pesquisa relacionados e ferramentas

Muitas das razões que motivaram esse trabalho, também motivaram vários projetos de pesquisa: a grande utilização do padrão MPI, sua interface simplificada e o alto desempenho das implementações MPI. Mesmo que esses trabalhos tenham diferentes abordagens, os trabalhos relacionados listado nas subseções seguintes buscam a resolução e problemas relacionados a computação em Grades para permitir uma utilização eficiente de MPI para desenvolver aplicações para Grades.

A maioria das implementações de MPI para grades são baseadas na implementação MPICH do padrão devido à sua organização modular. MPICH separa a interface de programação e a camada de comunicação, a qual é chamada através da ADL (acrônimo para *Abstract Device Interface*). Entre essas soluções, podemos citar PACX-MPI, MPICH-G2 e GridMPI. As próximas subseções apresentam cada um desses projetos e suas abordagens para permitir a utilização de MPI em grades computacionais.

A.3.1 PACX-MPI

PACX-MPI é desenvolvido no High Performance Computing Center of Stuttgart (HLRS). PACX-MPI inclui *daemons* em cada cluster, os quais são responsáveis pelo redirecionamento de mensagens entre os diferentes clusters. Os nós responsáveis pelo redirecionamento de mensagens são definidos e configurados previamente à execução para que sejam capazes de comunicar-se apesar de limitações de conexão, como firewalls.

PACX-MPI é implementado como uma biblioteca que fica entre a aplicação de usuários e a distribuição MPI instalada localmente nos clusters. Quando as aplicações enviam mensagens, estas são interceptadas pela biblioteca PACX-MPI que verifica a necessidade de contactar processos MPI executados em clusters distantes. Se as mensagens são enviadas no contexto de um cluster elas passam pela camada de comunicação. Senão, elas são enviadas através dos *daemons* PACX-MPI.

PACX-MPI definiu um sistema de identificação dos processos MPI no qual cada processo contém dois identificadores (ranks), um local e um global. A biblioteca PACX-MPI decide, automaticamente quando utilizar cada um desses identificadores de acordo com a topologia dos recursos e localização dos processos que se comunicam. A figura A.1 mostra como esses identificadores são escolhidos.

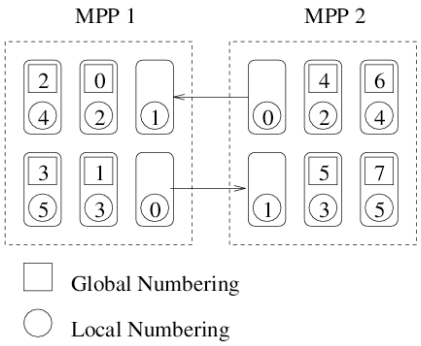

Figure A.1: Sistema de identificações dos processos PACX-MPI

PACX-MPI não define extensões ao padrão MPI. Portanto, aplicações ditas legadas podem ser executadas em Grades computacionais sem nenhuma modificação.

A.3.2 MPICH-G2

MPICH-G2 (KARONIS; TOONEN; FOSTER, 2003), desenvolvido no Computer Science Department of the Northern Illinois University em conjunto com a Argonne National Laboratory é uma solução completa baseada em MPICH que utiliza serviços do Globus Toolkit para permitir a execução de MPI em Grades computacionais, baseado no middleware Globus.

A camada mais baixa dessa implementação é composta por serviços Globus responsáveis pela alocação de recursos (GRAM), autenticação (GSI) e comunicação, quando a comunicação nativa não é possível (GlobusIO). Acima desses serviços fica a interface de programação MPICH, que por sua vez, é utilizada nas aplicações de usuários. A figura A.2 mostra a pilha de softwares envolvidos nesse processo.

As principais funcionalidades oferecidas por MPICH-G2 inclui a gestão de ambientes multi-domínio e a comunicação de alto desempenho em ambientes heterogêneos. MPICH-G2 também inclui, na definição de propriedades dos processos

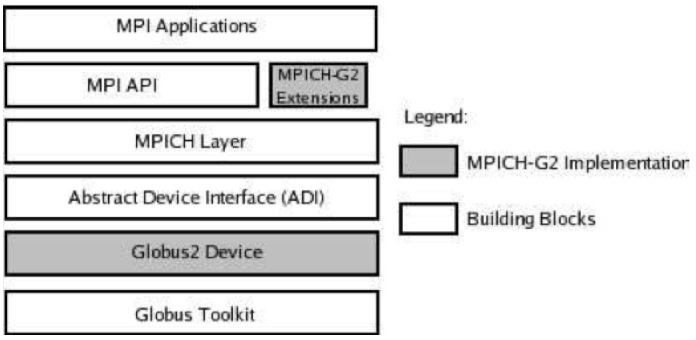

Figure A.2: Arquitetura MPICH-G2

MPI, informações sobre a localização de processos na topologia geral de recursos.

A.3.3 GridMPI

GridMPI (MATSUDA; KUDOH; ISHIKAWA, 2003) é outra implementação de MPI para Grades, desenvolvido pela University of Tokyo e o National Institute of Advanced Industrial Science and Technology (AIST). GridMPI é projetado para o desenvolvimento de aplicações de alto desempenho para Grades. GridMPI conecta múltiplos clusters distribuídos geograficamente para dar a impressão de um recurso computacional paralelo único.

Um dos principais objetivos de GridMPI é realizar comunicações globais de maneira eficiente através de otimizações que levam em conta a existência de conexões com latência e vazão heterogêneas. Além disso GridMPI esconde detalhes da topologia de rede envolvida na aplicação. Para tal, GridMPI oferece variações de algoritmos que implementam as comunicações coletivas em ambientes heterogêneos.

A respeito de desempenho, de acordo com os usuários de (MATSUDA; KUDOH; ISHIKAWA, 2003), GridMPI tem se mostrado a mais rápida entre as implementações do padrão MPI em ambientes heterogêneos.

A.4 Proposta de programação hierárquica orientada a Grids

Antes de mais nada, é importante ressaltar que no contexto desse trabalho, consideramos a hipótese de Grades computacionais são inerentemente organizadas hierarquicamente (p.ex. uma grade pode ser considerada um grupo de nós multi-core agrupados em nós de um cluster, que compões uma Infra-Estrutura de Grade maior entre múltiplos domínios administrativos). Uma conseqüência imediata dessa organização é que o desempenho da rede conectando os nós não é homogênea, levando a diferentes desempenhos de rede (latência e banda passante). Programadores são, então, fortemente aconselhados a priorizar comunicação local em detrimento de comunicações entre diferentes clusters. É importante citar também que esta estratégia, apesar de simples, é uma boa abordagem para melhorar o desempenho de aplicações (DONG; KARONIS; KARNIADAKIS, 2006).

Nesta seção, expomos as principais razões que levam à nossa abordagem. Em seguida, apresentamos uma API simplificada assim como alguns detalhes da nossa solução.

A.5 Especificação

Como apresentado previamente, a intenção é manter, ao máximo, o estilo de programação MPI. Para isso, propomos extensões de abstrações já existentes em MPI, como comunicadores, rank de processos e o suporte para comunicação e descoberta de topologias de tal forma que a complexidade associadas a Grades fique transparentes e a biblioteca proposta

A.5.1 Novos comunicadores MPI

Idealmente, consideramos um Grid como uma arquitetura em camadas:

- o primeiro e mais baixo nível (L1 por simplicidade) é caracterizado por cada processador de um nó;
- o segundo nível (L2) é representado por um nó, que potencialmente possui vários elementos L1;
- o terceiro nível (L3) consiste em um cluster. Tipicamente, cada cluster possui entre algumas dezenas e milhares de nós, normalmente interconectados por uma rede de alta velocidade;
- o quarto nível (L4) consiste de um nó de uma grade computacional, potencialmente composto por vários clusters geograficamente próximos. Esses cluster são tipicamente interconectados por uma rede dedicada de alto desempenho;
- o quinto nível (L5) representa o Grid, que consiste em um pequeno número de sítios geográficos L4 (< 50). Esses nós L4 são tipicamente conectados por uma rede WAN, e portanto apresentam maiores latências e uma banda passante reduzida.

Como o sistema operacional gerencia recursos nos níveis L1 e o padrão MPI no nível L2, introduzimos dois novos comunicadores para L3 e L4:

- comunicador `MPI_COMM_SITE`: contém referências a todos os processos em nós alocados no contexto de um site da grade;
- comunicador `MPI_COMM_GRID`: contém referência a todos processos no contexto da grade como um todo..

No protótipo desenvolvido no mestrado, não existe suporte a primitivas MPI como `MPI_Comm_Split`, `MPI_Comm_Merge` e `MPI_Comm_Dup`.

A figura A.3 mostra como os ranks são organizados em cada um desses comunicadores. Esta é uma forma bastante simples, mas adequada de identificar os processos MPI. Simples porque pode ser feito no momento de lançamento da aplicação, fornecendo uma visão global do a,ambiente. Conveniente porque a abstração dos comunicadores é mantida, o que torna fácil a utilização a programadores acostumados com o padrão MPI.

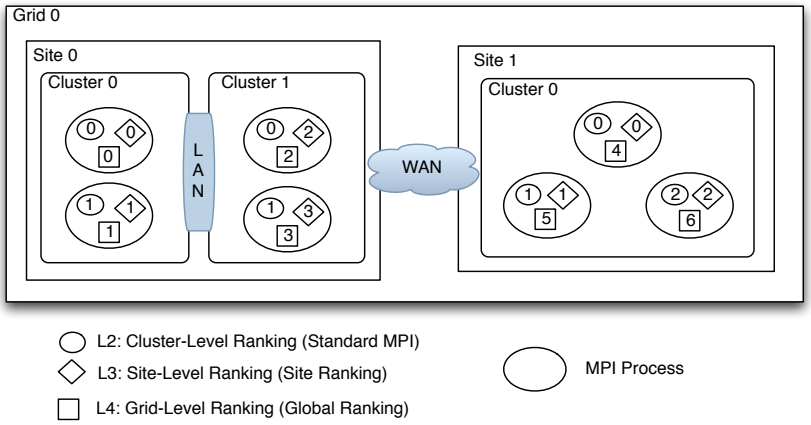

Figure A.3: Comunicadores Hierárquicos e Ranks

A.5.2 Novas primitivas MPI

Além da abstração dos comunicadores de mais alto nível, incluímos um grupo de primitivas principalmente para descoberta da topologia de processos. Para citar alguns exemplos:

- `MPI_ClusterInfo` e `MPI_SiteInfo` são estruturas de dados que contém informações sobre processos em um cluster ou site, respectivamente; `MPI_Comm_getMyClusterInfo (MPI_Comm comm, MPI_ClusterInfo *)` e `MPI_Comm_getMySiteInfo (MPI_Comm comm, MPI_SiteInfo *)` podem ser utilizados para obter as informações em relação aos processos que chamam essas primitivas;

- `MPI_COMM_SITE_GATEWAYS` e `MPI_COMM_GRID_GATEWAYS` são comunicadores especiais que agregam um nó em cada um dos clusters de um site ou dos sites da grade (normalmente o rank 0). A intenção desses comunicadores ;e de oferecer abstrações para tornar mais fácil a programação de aplicações hierárquicas;

- `MPI_Comm_getNeighborhood (int rank1, int rank2, MPI_Comm)` pode ser utilizada para obter a relação de vizinhança entre dois processos MPI ni contexto de um comunicador. Isto é, o numero de indireções necessárias para enviar uma mensagem entre os dois processos, dada a atual organização da grade.

- `MPI_Comm_translate(int in_rank, MPI_Comm comm_in, int* out_rank, MPI_Comm comm_out)`: cada processo possui diferentes ranks de acordo com o contexto de comunicação. Essa primitiva permide a tradução de um rank entre dois contextos diferentes.

Algumas primitivas MPI padrão continua válida com os novos comunicadores, como `MPI_Comm_rank` e `MPI_Comm_size`. Entretanto o protótipo atual só suporta as primitivas de comunicação mais utilizadas: `MPI_Send`, `MPI_ISend`, `MPI_Recv`, `MPI_IRecv`, `MPI_Bcast`, `MPI_Barrier` `MPI_Scatter`, `MPI_Scatterv`, `MPI_Gather` `MPI_Gatherv`, `MPI_Barrier` e `MPI_Gatherv`.

A.6 Suporte a aplicações inalteradas

Algumas aplicações, ditas `embarrassingly parallel` não necessitam muita comunicação entre processos. Além disso usuários podem estar interessados em executar aplicações sem modificação. Para esses casos, definimos uma macro C (#UNMODIFIED). Essa macro pode ser adicionada a qualquer código fonte MPI e o comunicador `MPI_COMM_WORLD` terá o mesmo comportamento de `MPI_COMM_GRID`.

A.7 Projeto e implementação do Runtime à Base de Componentes

Na seção precedente, introduzimos extensões ao padrão MPI para o desenvolvimento de algoritmos adaptados a ambientes de grades computacionais. Para que estas primitivas sejam efetivas, algumas funcionalidades são necessárias da parte do runtime:

- lançamento das aplicações em múltiplos clusters;
- suporte às primitivas que oferecem informação sobre a topologia dos recursos;
- suporte à comunicação inter-cluster.

A.8 Princípios

Uma série de princípios guia o projeto e implementação desse runtime. Esses princípios são importantes para melhor compreender escolhas de projeto e implementação desse runtime:

1. Usar comunicação nativa MPI o máximo possível;
2. Evitar indireções no processo de comunicação inter-cluster;
3. Tratamento hierárquico, principalmente das comunicações coletivas;
4. Problemas relacionados a utilização de Grades são transparentes para usuários.

A.9 Arquitetura de Software

O suporte às primitivas apresentadas previamente é baseado em uma arquitetura de software em camadas, organizados como na figura A.4. Esta figura apresenta as camadas de software presente em cada nó da Grade.

No topo se situa a aplicação final dos usuários, a qual acessa o runtime através da interface MPI e as extensões propostas. Nesse nível decide-se se a comunicação vai ser feita utilizando a pilha MPI ou o nosso runtime, o qual é executado por uma JVM. A camada mais abaixo é o sistema operacional que oferece suporte à comunicação de baixo nível.ss management and network.

As caixas em cinza indicam quais módulos foram produzidos no contexto desta dissertação.

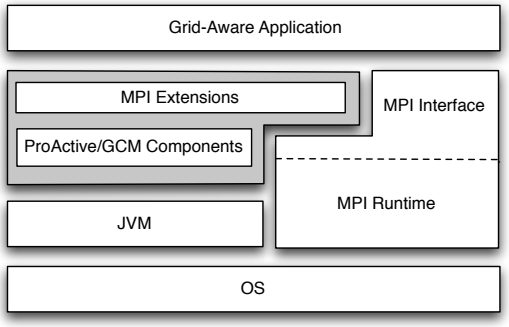

Figure A.4: Arquitetura do framework

A.10 Infra-Estrutura de Componentes

O principal objetivo do runtime é oferecer suporte à comunicação inter-cluster de maneira eficiente e transparente para usuários. Isso é alcançado através;es de uma infra-estrutura de componentes que modela uma hierarquia de recursos alocados para a execução de uma dada aplicação. Essa infra-estrutura é descentralizada e apenas existe durante a execução de aplicações, sem necessidade de instalação de software outra que uma distribuição MPI.

A.10.1 Componentes Básicos

Dois mecanismos são necessários à comunicação inter-cluster: (i) comunicação entre os processos nativos MPI e o ambiente Java e vice-versa e (ii) roteamento de mensagens através da rede. Cada uma dessas funcionalidades é oferecida por um componente diferente:

- componente **wrapper** (Figura A.5.a) é o componente mais elementar. Ele encapsula o processo MPI e é responsável pela comunicação MPI-Java. Isso inclui o encapsulamento de mensagens brutas em objetos Java incluindo informação que vai ser utilizada mais tarde no processo de roteamento.

  Componentes do tipo **wrapper** apresentam apenas uma interface servidora (*Srv*) e uma interface cliente *Clnt*, que são conectadas à um componente do tipo **clustering** que representa o cluster no qual o processo MPI embalado e o componente **wrapper** se encontram

- componente **clustering** (Figura A.5.b) é um componente genérico capaz de agrupar componentes de nível mais baixo. Nesse caso um componente **clustering** no nível 3 (L3) agrupa componentes L2 (componentes **wrapper**). Um componente **clustering** L4 agrupa componentes **clustering** L3 e assim por diante até que um componente **clustering** agrupe todos os componentes da grade.

  Componentes **clustering** apresentam uma interface servidor *Srv* que é responsável por receber requisições vindas de ambos os níveis, superior e inferior. Do lado cliente, componentes **clustering** apresentam duas interfaces cliente: uma para enviar mensagens ao nível superior (interface *GoUP*) e outra para enviar mensagens ao nível inferior (interface *GoDown*). Essas interfaces tornam possível aos componentes **clustering** de se comunicar com múltiplos

componentes em paralelo e essa funcionalidade é utilizada na prática para implementam a maioria das primitivas de comunicação coletiva MPI.

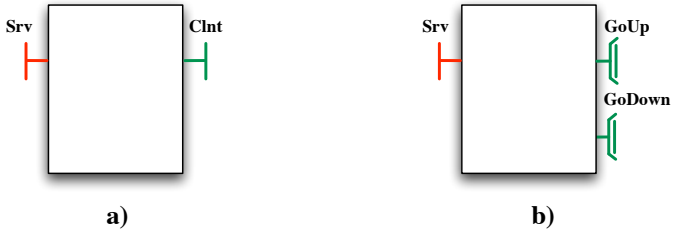

Figure A.5: a) Componente Wrapper b) Componente Clustering

Próximas seções mostram como estes componentes são acoplados para formar a infra-estrutura necessária e como as comunicações inter-cluster acontecem.

A.10.2 Mapeando Recursos como Componentes

Como discutido previamente, a infra-estrutura de componentes é modelada para refletir a organização de recursos, de tal forma que qualquer processo possa se comunicar com qualquer processo, mesmo que indiretamente. Esse processo acontece em momento de lançamento da aplicação na qual componentes *wrapping* são lançados em cada nó onde um processo MPI vai ser executado e recebem um identificado único, formado pelo identificador do nível mais baixo da hierarquia e o identificador MPI (rank) do processo correspondente.

A figura A.6 mostra um deployment simples em dois sites com dois e um cluster respectivamente e sete nós para processamento. Cada nível recebe um identificador, que se soma ao identificador do nível superior.

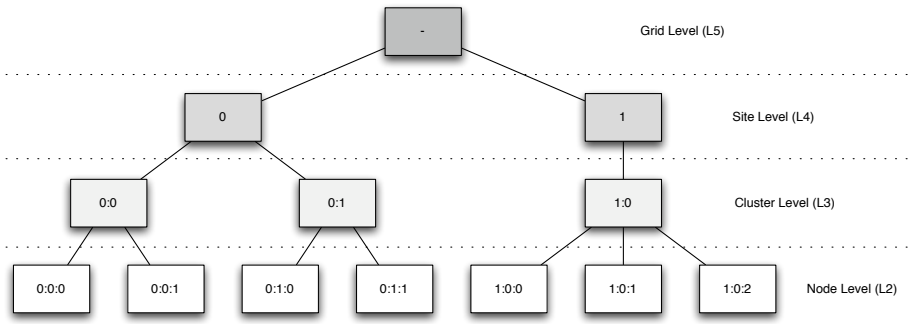

Figure A.6: Identificação dos Recursos e Componentes

A figura A.7 mostra os componentes responsáveis pela encapsulamento e comunicação inter-cluster. Por questões de clareza, apenas os componentes lançados na parte esquerda do grid hipotético (site *'0'* of the figure A.6) são representados.

Nessa organização, existe um componente *wrapper* (L2, em branco) para cada processo MPI, um componente *clustering* para cada cluster e um componente *clustering* L4 para todo o site. De maneira geral, componentes *clustering* são lançados nos frontends dos clusters (por padrão o nó 0, ou o n;o que é definido manualmente). Na configuração padrão, a comunicação entre os componentes é feita utilizando o protocolo RMI, mas outros protocolos podem ser utilizados (rmissh, ibis, http or even soap), especialmente quando o protocolo RMI não é autorizado.

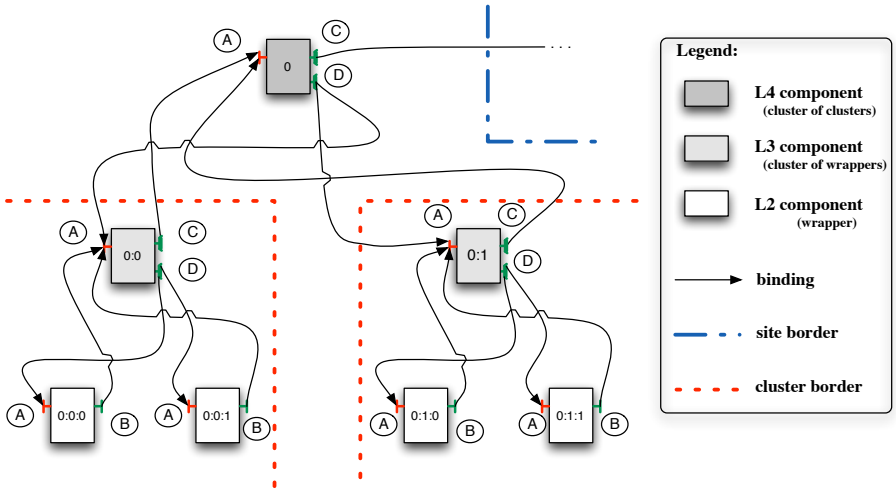

Figure A.7: Organização dos Componentes

A.11 Message Routing over the Grid

Depois do lançamento do ambiente, uma visão global pode ser construída e cada processo possui a visão local da topologia através do mapeamento entre comunicadores e os identificadores únicos. A tabela seguinte mostra no exemplo, a tabela de identificadores que estaria estocada nos nós *0:1:0* e *0:1:1*.

Ranks \ Comms	MPI_COMM_CLUSTER	MPI_COMM_SITE	MPI_COMM_GRID
0	0:0:0	0:0:0	0:0:0
1	0:0:1	0:0:1	0:0:1
2	-	0:1:0	0:1:0
3	-	0:1:1	0:1:1
4	-	-	1:0:0
5	-	-	1:0:1
6	-	-	1:0:2

Table A.1: Mapeamento de identificadores de componentes para comunicadores hierárquicos

Baseado nessa tabela, decisões de roteamento são domadas e as mensagens ou seguem a comunicação MPI (se estiverem no mesmo cluster), ou a comunicação através dos componentes.

A.11.1 Mensagens ponto-a-ponto

Devido à existência dos identificadores únicos, o roteamento mensagens ponto-a-ponto é simples. asta a comparação do identificador destino com o local. Se os sois valores iniciais são idênticos, a mensagem é enviada através de uma comunicação simples `MPI_Send`. Senão, o componente *wrapper* se encarregará da comunicação.

A comunicação por parte de componentes *wrappers* ocorre em duas etapas:

1. Obtenção da referência do componente *wrapper* remoto através da interface *GoUp*.

2. Envio de mensagem de forma direta ao componente *wrapper* equivalente ao processo de destino.

No caso de impossibilidade de comunicação direta, a mensagem é enviada através da infra-estrutura de componentes até que ela chega ao seu destino. No processo receptor, o componente *wrapper* verifica a origem da mensagem e passa a mesma ao processo que executa na mesma máquina.

A.11.2 Collective Messages

Da mesma forma que o padrão MPI, primitivas de comunicação coletiva ocorrem no contexto de um dado comunicador e todos os processos deste comunicador devem participar. O roteamento de comunicações coletivas é mais complexo do que comunicações ponto-a-ponto e depende to tipo de comunicação coletiva.

- Broadcast e Scatter: componentes *wrappers* correspondentes aos processos envolvidos em uma comunicação broadcast tem dois comportamentos distintos. Componentes não-*root* apenas esperam pela mensagem e o componente *root* inclui na mensagem informações sobre o tipo de operação e seu identificador e envia aos níveis superiores através de sua interface cliente *GoUp*. Essa mensagem é retransmitida até que chegue ao nível correspondente ao comunicador no qual a comunicação ocorre. Uma vez que a mensagem chega nesse componente *clustering*, ele envia a mensagem a todos os processos em paralelo através de sua interface cliente *GoDown*. O recebimento da notificação equivalente à essa mensagem no *wrapper* root da operação desbloqueia o processo root.

  Na comunicação do tipo scatter, o mesmo processo ocorre, mas ao invés de enviar toda a mensagem a cada nível, somente as partes da mensagem a serem transmitidas são transmitidas.

- Gather e Reduce: no caso de comunicações do tipo gather, o comportamento e exatamente o oposto. O componente *wrapper* do processo root espera simplesmente por uma mensagem e todos os demais processos (e por conseqüência os componentes) enviam através da interface cliente *GoUp* sua parte dos dados. O componente *clustering* de mais alto nível do comunicador fica bloqueado até que todas as mensagens chegam, ordenando as mensagens à medida que as mesmas chegam, baseado nas meta-informações contidas nas mensagens, que inclui a origem de cada mensagem. A mensagem completa é, então enviada ao componente root e uma mensagem de *acknowledgement* é enviada aos demais processos para que eles desbloqueiem.

  O mesmo ocorre com primitivas do tipo *Reduce*, mas os componentes *clustering* intermediários reduzem os dados, se possível, antes de passarem aos próximos componentes na cadeia.

- Barrier: no caso de operações do tipo barrier, o comportamento dos nós é similar ao *broadcast*, cada nó envia uma mensagem até o componente *clustering*

de mais alto nível e espera por uma confirmação de recebimento. O comportamento do componente *clustering* é similar ao comportamento de *gathering*, uma vez que ele recebe todas as mensagens ele envia a confirmação para desbloquear todos os processos do comunicador.

Cada componente do tipo *clustering* possui uma fila para armazenar mensagens coletivas. O tratamento de mensagens obedece a um ordenamento FIFO não-bloqueante de forma que o ordenamento de mensagens definido pelo padrão MPI seja obedecido. A chegada de uma mensagem relativa a uma chamada coletiva que depende da recepção de múltiplas mensagens desencadeia a criação de uma entrada na fila para tratar essa chamada, sendo que quando o total de mensagem é recebida, a mensagem é despachada, caracterizando o comportamento não-bloqueante.

A.12 Avaliação

Os resultados obtidos na avaliação do runtime desenvolvido no contexto dessa dissertação mostram que a sobrecarga gerada pela camada de componentes é negligível. Entretanto, em infra-estruturas multi-cluster, eles impactam o desempenho geral de aplicações. Este fato aumenta a importância das otimizações do runtime que garantem uma comunicação direta entre componentes sempre que possível. O tratamento hierárquico das comunicações coletivas também;em mostrou ganhos significativos de desempenho.

Os resultados obtidos em experimentos mostraram que aplicações paralelas do tipo *non-embarrassingly* devem ser *gridificadas* para obter vantagens da utilização de recursos em múltiplos clusters, de tal forma que possam apresentar uma boa escalabilidade em recursos de Grade.

Uma análise qualitativa mostrou que, embora com abordagens bastante diferentes, nossa solução apresenta a maioria das funcionalidades oferecidas pelos trabalhos relacionados. Isto mostra que o caráter geral da nossa solução, a qual pode ser utilizada em diferentes contextos de grades para o desenvolvimento de diferentes tipos de aplicações paralelas.

Mais informações sobre a avaliação do trabalho e resultados podem ser encontrados no capítulo 7.

A.13 Conclusão

A natureza heterogênea e hierárquica de grades computacionais aumenta a importância de problemas e, principalmente, a necessidade de modelos de programação adequados. Devido a alta aceitação do modelo de passagem de mensagens e do padrão MPI como o paradigma padrão para desenvolver aplicações de alto desempenho, a idéia de usar MPI em grades computacionais tem sido objeto de pesquisa atualmente.

Até agora, a principal abordagem para MPI em grades tem sido a utilização de aplicações não modificadas. De um lado isso pode facilitar a utilização de aplicações existentes, mas por outro lado, algoritmos paralelos devem refletir a topologia de recursos para apresentarem um bom desempenho e escalabilidade. Apesar desse requisito, identificamos uma falta de mecanismos e abstrações que permitissem o desenvolvimento de aplicações MPI hierárquicas para grids acompanhado de outros modelos mais apropriados à ambientes de grade, como o de componentes .

Nesse contexto, propomos um modelo híbrido através de extensões ao padrão MPI que permitem a utilização do modelo de passagem de mensagens de forma hierárquica assim como um framework baseado em componentes suportando as funcionalidades introduzidas. Esse protótipo fez uso do middleware para grades ProActive e de seu suporte ao lançamento de aplicações em grades, encapsulamento de código nativo e a implementação de GCM.

Or resultados de desempenho obtidos mostram que a sobrecarga gerada pelo uso de componentes não é negligível, mas dentro do esperado. Pode-se, inclusive, esperar que os benefícios em termos de desempenho para aplicações sejam superiores à perda de desempenho. Entretanto, aplicações, especialmente *non-embarrassingly* devem levar em conta a topologia de recursos.

APPENDIX B DEPLOYMENT SCHEMA

The following piece of XML Schema (.xsd) defines the dependent process (presented in the section 6.3)

```
    .........
<!-- mpiProcessType -->
<xs:complexType name="mpiProcessType" mixed="true">
    <xs:all>
        <xs:element minOccurs="1" ref="commandPath" />
        <xs:element minOccurs="1" name="mpiOptions"
            type="mpiOptionsType" />
    </xs:all>
    <xs:attribute
        fixed="org.objectweb.proactive.core.process.mpi.MPIDependentProcess"
        name="class" type="xs:string" use="required" />
    <xs:attribute name="mpiFileName" type="xs:string"
        use="required" />
    <xs:attribute name="hostsFileName" type="xs:string"
        use="optional" />
    <xs:attribute name="mpiCommandOptions" type="xs:string"
        use="optional" />
</xs:complexType>

<!--mpiOptions-->
<xs:complexType name="mpiOptionsType" mixed="true">
    <xs:all>
        <xs:element minOccurs="1" name="processNumber"
            type="PosintOrVariableType" />
        <xs:element minOccurs="0" name="nolocal"
            type="TextOrVariableType" />
        <xs:element minOccurs="1" name="localRelativePath"
            type="FilePathType" />
        <xs:element minOccurs="0" name="remoteAbsolutePath"
            type="FilePathType" />
    </xs:all>
</xs:complexType>
<!-- end mpiOptions-->
<!-- end of mpiProcessType -->
```

```
<!-- dependentProcessSequenceType -->
<xs:complexType name="dependentProcessSequenceType" mixed="true">
    <xs:sequence>
        <xs:choice minOccurs="1">
            <xs:element name="serviceReference"
                type="ServiceReferenceType" />
            <xs:element ref="processReference" />
        </xs:choice>
        <xs:element maxOccurs="unbounded" ref="processReference" />
    </xs:sequence>
    <xs:attribute
        fixed="org.objectweb.proactive.core.process.DependentListProcess"
        name="class" type="xs:string" use="required" />
</xs:complexType>
<!-- end of dependentProcessSequenceType -->


    .........
```

APPENDIX C FLAT MONTECARLO CODE SAMPLE

```
...
MPI_Init (&argc, &argv); /* starts MPI */
MPI_Comm_rank (MPI_COMM_GRID, &rank); /* get current process id */

if(rank != 0){
   /*pi calculation in all nodes but the coordinator*/
}

/*reduce on node 0 of grid the average of results*/
MPI_Reduce(sendbuff,recvbuff,1,MPI_DOUBLE,MPI_AVG,
                  0,MPI_COMM_GRID);
MPI_Finalize();
...
```

APPENDIX D HIERARCHICAL MONTECARLO CODE SAMPLE

```
#include<pampi.h>
...
MPI_Init (&argc, &argv); /* starts MPI */

/* get current process grid_id */
MPI_Comm_rank (MPI_COMM_GRID, &grid_rank);

/* get current process gateway_id */
MPI_Comm_rank (MPI_COMM_GRID_GATEWAYS, &gateway_rank);

...

if(grid_rank != 0 && gateway_rank < 0){
    /*pi calculation in all nodes but the coordinators*/
}

/*reduce first on each site, node 0 is also the gateway*/
MPI_Reduce(sendbuff,recvbuff,1,MPI_DOUBLE,
      MPI_AVG,0,MPI_COMM_SITE);


/*reduce on node 0 of grid the average of results in sites*/
if(gateway_rank >= 0){
MPI_Reduce(recvbuff,result,1,MPI_DOUBLE,MPI_AVG,
                  0,MPI_COMM_GRID_GATEWAYS);
}
...
MPI_Finalize();
...
```

APPENDIX E FLAT MERGESORT CODE SAMPLE

```
#include<pampi.h>
...
#define UNMODIFIED 1
...
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
MPI_Comm_size(MPI_COMM_WORLD,&p);
...
if(id==0)
{
/*bradcast buffer size*/
    MPI_Bcast(&s,1,MPI_INT,0,MPI_COMM_WORLD);

    /*scatter data to all the processes*/
MPI_Scatter(data,s,MPI_INT,chunk,s,MPI_INT,0,MPI_COMM_WORLD);

for(i=0; i<(p-1); i++){
   /*receive ordered chunk*/
MPI_Recv(temp_chunk, s, MPI_INT, MPI_ANY_SOURCE,
      0, MPI_COMM_WORLD, &stat);

/*merge received chunk on local buffer*/
}
...
}else{
    /*Receive buffer size*/
MPI_Bcast(&s,1,MPI_INT,0,MPI_COMM_WORLD);

/**receive data from scatter/
MPI_Scatter(data,s,MPI_INT,chunk,s,MPI_INT,0,MPI_COMM_WORLD);

/*order chunk locally*/

/*send ordered chunk to master*/
MPI_Send(chunk,s,MPI_INT,0,0,MPI_COMM_WORLD);
}
```

```
...

MPI_Finalize();
```

APPENDIX F HIERARCHICAL MERGESORT CODE SAMPLE

```
#include<pampi.h>

...
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_GRID_GATEWAYS, &grid_gateway_rank);
MPI_Comm_size(MPI_COMM_GRID_GATEWAYS, &site_num);
MPI_Comm_rank(MPI_COMM_SITE_GATEWAYS, &site_gateway_rank);
MPI_Comm_size(MPI_COMM_SITE_GATEWAYS, &cluster_num);

...
/*grid coordinator*/
if(grid_gateway_rank == 0)
{
/*bradcast buffer size to sites*/
    MPI_Bcast(&s1,1,MPI_INT,0,MPI_COMM_GRID_GATEWAYS);

    /*scatter data to sites*/
MPI_Scatter(data,s1,MPI_INT,chunk,s1,MPI_INT,0,MPI_GRID_GATEWAYS);

    /*receive ordered chunk*/
for(i=0; i<(site_num-1); i++){
MPI_Recv(temp_chunk, s, MPI_INT, MPI_ANY_SOURCE,
      0, MPI_GRID_GATEWAYS, &stat);

/*merge received chunk on local buffer*/
}
...


}else
/*site coordinator*/
if (grid_gateway_rank >= 0)
{
    /*Receive buffer size*/
```

```
MPI_Bcast(&s1,1,MPI_INT,0,MPI_COMM_GRID_GATEWAYS);

    /*bradcast buffer size to clusters*/
    MPI_Bcast(&s2,1,MPI_INT,0,MPI_COMM_SITE_GATEWAYS);

    /*receive data on sites*/
MPI_Scatter(data,s1,MPI_INT,chunk,s1,MPI_INT,0,MPI_GRID_GATEWAYS);
...
/*scatter data to clusters*/
MPI_Scatter(data,s2,MPI_INT,chunk,s2,MPI_INT,0,MPI_SITE_GATEWAYS);

for(i=0; i<(cluster_num-1); i++){
   /*receive ordered chunk*/
MPI_Recv(temp_chunk, s2, MPI_INT, MPI_ANY_SOURCE,
      0, MPI_SITE_GATEWAYS, &stat);

/*merge received chunk on local buffer*/
}

MPI_Send(chunk,s1,MPI_INT,0,0,MPI_GRID_GATEWAYS);
...

} else
/*cluster coordinator*/
if (cluster_gateway_rank >= 0)
{
    /*Receive buffer size*/
MPI_Bcast(&s2,1,MPI_INT,0,MPI_COMM_SITE_GATEWAYS);

    /*bradcast buffer size to clusters*/
    MPI_Bcast(&s3,1,MPI_INT,0,MPI_COMM_CLUSTER);

    /*receive data on sites*/
MPI_Scatter(data,s2,MPI_INT,chunk,s2,MPI_INT,0,MPI_SITE_GATEWAYS);
...
/*scatter data to computing nodes*/
MPI_Scatter(data,s3,MPI_INT,chunk,s3,MPI_INT,0,MPI_COM_CLUSTER);

for(i=0; i<(cluster_num-1); i++){
   /*receive ordered chunk*/
MPI_Recv(temp_chunk, s3, MPI_INT, MPI_ANY_SOURCE,
      0, MPI_COMM_CLUSTERS, &stat);

/*merge received chunk on local buffer*/
}

MPI_Send(chunk,s2,MPI_INT,0,0,MPI_SITE_GATEWAYS);
...
```

```
}
/*computing nodes*/
else{
    /*Receive buffer size*/
MPI_Bcast(&s3,1,MPI_INT,0,MPI_COMM_WORLD);

/**receive data from scatter/
MPI_Scatter(data,s3,MPI_INT,chunk,s3,MPI_INT,0,MPI_COMM_CLUSTER);

/*order chunk locally*/

/*send ordered chunk to master*/
MPI_Send(chunk,s3,MPI_INT,0,0,MPI_COMM_CLUSTER);
}
...

MPI_Finalize();
```

REFERENCES

ALDINUCCI, M.; CAMPA, S.; CIULLO, P.; COPPOLA, M.; DANELUTTO, M.; PESCIULLESI, P.; RAVAZZOLO, R.; TORQUATI, M.; VANNESCHI, M.; ZOCCOLO, C. A framework for experimenting with structure parallel programming environment design. In: PARALLEL COMPUTING: SOFTWARE TECHNOLOGY, ALGORITHMS, ARCHITECTURES AND APPLICATIONS, PARCO 2003, 2004, Dresden, Germany. **Anais...** Elsevier, 2004. p.617–624. (Advances in Parallel Computing, v.13).

AUMAGE, O.; MERCIER, G. **MPICH/MADIII**: a cluster of clusters enabled mpi implementation. 2003.

AUMAGE, O.; MERCIER, G.; NAMYST, R. MPICH-Madeleine: a True Multi-Protocol MPI for High-Performance Networks. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2001), 15., 2001, San Francisco. **Proceedings...** [S.l.: s.n.], 2001. p.51.

BADUEL, L.; BAUDE, F.; CAROMEL, D.; CONTES, A.; HUET, F.; MOREL, M.; QUILICI, R. **Grid Computing**: software environments and tools. [S.l.]: Springer-Verlag, 2006.

BAKER, M.; BUYYA, R.; LAFORENZA, D. **Grids and Grid technologies for wide-area distributed computing**. 2002.

BAUDE, F.; CAROMEL, D.; HENRIO, L.; MOREL, M. Collective Interfaces for Distributed Components. In: CCGRID 2007: 7TH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2005. **Anais...** [S.l.: s.n.], 2005. p.599–610.

BEISEL, T.; GABRIEL, E.; RESCH, M. An Extension to MPI for Distributed Computing on MPPs. In: PVM/MPI, 1997. **Anais...** [S.l.: s.n.], 1997. p.75–82.

BERMAN, F.; FOX, G.; HEY, T. **Grid computing**: making the global infrastructure a reality. [S.l.]: Wiley, 2003.

BOTE-LORENZO, M. L.; DIMITRIADIS, Y. A.; GOMEZ-SANCHEZ, E. G. Grid Characteristics and Uses: a grid definition. **European Across Grids Conference (ACG/03)**, Piscataway, NJ, USA, v.1, n.5, p.291–298, 2004.

BRON, C. Merge Sort Algorithm. **Commun. ACM**, [S.l.], v.15, n.5, p.357–358, May 1972.

BRUNETON, E.; COUPAYE, T.; STEFANI, J.-B. **The Fractal Component Model Specification**. 2004.

CERA, M. C.; PEZZI, G. P.; PILLA, M. L.; MAILLARD, N.; NAVAUX, P. O. A. Scheduling dynamically spawned processes in MPI-2. In: JSSPP'06: PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 2007, Berlin, Heidelberg. **Anais...** Springer-Verlag, 2007. p.33–46.

COLE, M. I. **Algorithmic Skeletons**: structured management of parallel computation. [S.l.]: Pitman, 1989. (Research Monographs in Parallel and Distributed Computing).

COREGRID Network of Excellence. 2007.

DONG, S.; KARONIS, N. T.; KARNIADAKIS, G. E. Grid solutions for biological and physical cross-site simulations on the TeraGrid. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2006), 20., 2006. **Proceedings...** IEEE, 2006.

DUNNWEBER, J.; GORLATCH, S. HOC-SA: a grid service architecture for higher-order components. In: SCC '04: PROCEEDINGS OF THE 2004 IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.288–294.

FAGG, G. E.; BUKOVSKY, A.; DONGARRA, J. J. HARNESS and fault tolerant MPI. **Parallel Computing**, [S.l.], v.27, n.11, p.1479–1495, Oct. 2001.

FOSTER, I. What is the Grid? A Three Point Checklist. **GRID Today Magazine**, [S.l.], v.1, n.6, July 2002.

FOSTER, I.; KARONIS, N. A Grid-Enabled MPI: message passing in heterogeneous distributed computing systems. In: **Proceedings of SC'98**. [S.l.]: ACM Press, 1998.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid**: blueprint for a future computing infrastructure. [S.l.]: MORGAN-KAUFMANN, 1999.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. [S.l.]: Morgan Kaufmann Publishers, 2003. ISBN: 1-55860-933-4.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus Approach to Integrating Multithreading and Communication. **Journal of Parallel and Distributed Computing**, [S.l.], v.37, n.1, p.70–82, 1996.

FOX, G. Message Passing: from parallel computing to the grid. **Computing in Science and Engg.**, Piscataway, NJ, USA, v.4, n.5, p.70–73, 2002.

GABRIEL, E.; RESCH, M.; BEISEL, T.; KELLER, R. Distributed Computing in a Heterogeneous Computing Environment. In: PVM/MPI USERS' GROUP MEETING, 1998. **Anais...** [S.l.: s.n.], 1998. p.180–187.

GEOFFRAY, P.; PRYLLI, L.; TOURANCHEAU, B. BIP-SMP: high performance message passing over a cluster of commodity SMPs. In: SC'99, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.142–151.

GRID Component Model Specification. 2007.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Computing**, [S.l.], v.22, n.6, p.789–828, Sept. 1996.

JAGANNADHAM, D.; RAMACHANDRAN, V.; KUMAR, H. Java2 distributed application development (Socket, RMI, Servlet, CORBA) approaches, XML-RPC and web services functional analysis and performance comparison. **Communications and Information Technologies, 2007. ISCIT '07. International Symposium on**, [S.l.], p.1337–1342, Oct. 2007.

KARONIS, N.; TOONEN, B.; FOSTER, I. MPICH-G2: a grid-enabled implementation of the message passing interface. **Journal of Parallel and Distributed Computing (JPDC)**, [S.l.], v.63, n.5, p.551–563, may 2003.

KELLER, R.; KRAMMER, B.; MULLER, M. S.; RESCH, M. M.; GABRIEL, E. **MPI Development Tools and Applications for the Grid**. 2003.

KURZYNIEC, D.; HWANG, P.; SUNDERAM, V. Failure Resilient Heterogeneous Parallel Computing Across Multidomain Clusters. **International Journal f High Performance Computing Applications (IJHPCA), Special Issue: Best Papers of EuroPVM/MPI 2004**, [S.l.], 2005. (accepted).

MATSUDA, M.; KUDOH, T.; ISHIKAWA, Y. Evaluation of MPI Implementations on Grid-connected Clusters using an Emulated WAN Environment. In: CCGRID '03: PROCEEDINGS ..., 2003, Washington, DC, USA. **Anais...** IEEE Computer Society, 2003. p.10.

MOREL, M. **Components for Grid Computing**. 2006. Tese (Doutorado em Ciência da Computação) — Universite de Nice - Sophia Antipolis.

MPI Forum. **MPI**: a message-passing-interface standard. [S.l.]: MPI Forum, 1994.

MPI Forum. **MPI-2.0**: extensions to the message-passing interface. [S.l.]: MPI Forum, 1997.

PEZZI, G. P.; CERA, M. C.; MATHIAS, E.; MAILLARD, N.; NAVAUX, P. O. A. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. **Computer Architecture and High Performance Computing, Symposium on**, Los Alamitos, CA, USA, v.0, p.247–254, 2007.

SUNDERAM, V. S. PVM: a framework for parallel distributed computing. In: 1990. **Anais...** [S.l.: s.n.], 1990. v.2, n.4, p.315–339.

SZYPERSKI, C.; PFIZER, U. **WCOP96 Workshop Report**. p127-130, Worskshop Reader ECOOP96.

WILKINSON, B.; ALLEN, M. **Parallell Programming**. New Jersey: Prentice Hall, 1999. p.38–81.

YU, J.; BUYYA, R. A taxonomy of scientific workflow systems for grid computing. **SIGMOD Rec.**, New York, NY, USA, v.34, n.3, p.44–49, 2005.