Master RSD

Hierarchical Message Passing through Grid Components and the ProActive Platform

Elton Mathias

Projet OASIS, INRIA, CNRS-I3S, Université de Nice Sophia-Antipolis
Elton.Mathias@inria.fr

Encadrant : Françoise Baude

Fev-Juin 2007
5 Evaluation

5.1 Experimental Environment .............................................. 34
5.2 Benchmarks .............................................................. 34
5.3 Experiments ............................................................... 35
  5.3.1 Applications ......................................................... 36
  5.3.2 Results ............................................................. 39
5.4 Comparison with existing tools ......................................... 40

6 Conclusions and Perspective .............................................. 43
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>PACX-MPI Architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>PACX-MPI Rank Scheme</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>MPICH-G2 Architecture</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Startup of MPICH-G2</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>MPICH/Madeleine Architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>H₂O MPI Architecture: simple example</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchical Communicators and ranks</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecture of the Framework</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Components Identification</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>a. Wrapper component; b. Clustering component</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Assembly of components</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>single cluster point-to-point communication performance</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>cross-site point-to-point communication performance</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance of collective communication within a multi-site grid environment</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>Pi computation Scheme</td>
<td>36</td>
</tr>
<tr>
<td>5.5</td>
<td>Mergesort example</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>Flat Mergesort Scheme</td>
<td>38</td>
</tr>
<tr>
<td>5.7</td>
<td>Hierarchical Mergesort Scheme</td>
<td>38</td>
</tr>
<tr>
<td>5.8</td>
<td>Poisson sliced data partition</td>
<td>39</td>
</tr>
<tr>
<td>5.9</td>
<td>Monte Carlo Times</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>Monte Carlo Speedup</td>
<td>40</td>
</tr>
<tr>
<td>5.11</td>
<td>Flat Mergesort (i) Times</td>
<td>41</td>
</tr>
<tr>
<td>5.12</td>
<td>Hierarchical Mergesort (ii) Times</td>
<td>41</td>
</tr>
<tr>
<td>5.13</td>
<td>Comparison between flat and hierarchical ProActiveMPI versions</td>
<td>41</td>
</tr>
<tr>
<td>5.14</td>
<td>Flat Mergesort (i) Speedup</td>
<td>41</td>
</tr>
<tr>
<td>5.15</td>
<td>Hierarchical Mergesort (ii) Speedup</td>
<td>41</td>
</tr>
</tbody>
</table>
List of Tables

4.1  Mapping of ranks within hierarchical communicators to component identifiers . . . .   31
5.1  Comparison of the developed prototype with related tools . . . . . . . . . . . . . . . . 42
Chapter 1

Introduction

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

Research efforts first focused on the access to physical resources by providing tools for the search, reservation and allocation of resources, and for the construction of virtual organizations. Accessing a grid infrastructure is a first necessary step for taking advantage of the resources of the grids, and the second step is to offer adequate development models and environments (frameworks). A new research area therefore emerged, which focused on programming models and tools for efficiently programming applications which could be 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 [1],

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 standard with a large number of available libraries and legacy applications. For this reason, the usage of the well known and accepted MPI to develop grid applications has being recently investigated in research and industry.

While not a high level programming model by any means, the message passing model lacks dynamicity and abstractions to program grid applications. Indeed, the MPI standard addresses cluster environments, not having primitives adapted to program multi-site grid environments, that are inherently hierarchical. Contrary to message-passing, a component-based model encompasses most of the programming models proposed to grid programming [2] (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 applications.
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 model that combines the high performance and high acceptability of the MPI standard with the flexibility of the component-based programming model. This approach intends to meet grid programming requirements, offering MPI programmers a way to develop their applications/algorithms in a grid-aware hierarchical manner, yet taking profit of legacy high-performance codes.

Our solution relies on the addition of new MPI communicators, modeling the hierarchical structure of multi-site grid environments and a related API, that may offer an abstraction 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 these primitives, in a transparent way for MPI programmers.

In order to do so, the implementation will be based on the ProActive platform, which offers the adequate support to deploy and execute MPI applications and also offers a pre-prototype implementation of the CoreGRID [3] Grid Component Model (GCM) [4].

The main contributions of this work includes:

1. The definition of extensions to the MPI standard, addressing hierarchical communication;
2. a support to easily deploy SPMD MPI applications in grids;
3. a component-based message passing framework that expresses, in components, the MPI structure and communication patterns;
Chapter 2

Positioning and State of The Art

This chapter presents our positioning regarding to grid definition, characteristics and issues. In order to understand the approach of using message passing in grid environments, we analyse the main programming models for development of grid applications. After, we discuss the MPI standard and its applicability to grid computing and present some related works.

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 [5]. 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 [6].

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.

Such definition includes most of the characteristics of a grid environment, that are summarized on the next section.

Some authors consider this definition slightly abstract [7] and use a more specific definition. 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.
2.1.2 Characteristics

Although there exist several different definitions, a set of characteristics are common [8]:

- 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 impact in performance that is result of it;

- Heterogeneity: a grid hosts both software and hardware resources that can be very varied ranging from data, files, software components or programs to sensors, scientific instruments, display devices, personal digital organizers, computers, super-computers and networks.

- Resource sharing: grid resources usually belong to many different organizations that allow other organizations to access them. For that reason, the collection of resources can be seen as a great shared resource. This assumption lead to necessity of load balancing schemes and resource discovery mechanisms;

- 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 of taking into account all 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 [9], namely: portability and interoperability, security, fault tolerance, performance and the need of an adequate programming model.

Portability, Interoperability and Adaptingty Current high-level languages allow code to be processor-independent. In this sense, grid programming tools should also enable the applications to have the same portability [9]. This can means architecture independence in the sense of languages interpreted by virtual machines (namely, Java), but it also can mean the ability to use different services at different locations with equivalent functionality. Such portability is a necessary pre-requisite for coping with dynamic, heterogeneous configuration.

Another important point is the idea of interoperability, that stands in the notion of the capability of two or more components or component 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) [6].

Related to portability and interoperability, another important issue that arises from the inherent grid characteristics is the 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.
**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) [6], 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.

**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.

**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.

**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 [6] and, for this reason, many programming models have been proposed and adapted last years. The following 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 applications.
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 and 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 have a more complex usage [10]. 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 leverages issues like hierarchical topology, dynamicity and heterogeneity inhibits he usage of message passing in grid environments. However, the large amount of legacy applications and also the high acceptance of this model by the developer communities have created, recently, a great interest on the application of this model to grid computing.

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 [11], MPICH-Madeleine [12, 13], MPICH-G2 [14] and H$_2$O MPI. More details about these projects can be found in the section 2.3.

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 end-points. 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 discover remote procedure names, associating them with remote stub interface information at run-time.

Worthy to notice that this programming model by itself does not address directly 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 a widely used programming approach. Distributed communications between objects are easily realized 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 JavaSymphony, 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 [15], 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 3.1.

2.1.4.4 Skeleton Model

Skeletons are high-level and parametrized algorithmic patterns, introduced by Cole et. al [16]. 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) [17] 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. HOC-SA [18] 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) [2].

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, 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 [1]. However, 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 [19], 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.

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 contrac-
tually specified interfaces and explicit context dependencies only. Besides, a software component can be deployed independently and be subject to composition by third parties [20].

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 actually coordinated by the CCA Forum defines a component model especially for scientific applications. The model primarily addresses the heterogeneity and the separation of interface and implementation. 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 3.4.

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 the usage of MPI in actual grid infrastructures.

2.2.1 The MPI Standardization

The MPI standardization effort [21], 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 [22] 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 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 [21]:

- 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 or doing operations (merge, split, ...) with 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 enables the creation of communicators and assignment of ranks to process 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 a graph of processes;

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: mechanisms to use dynamic resources, co-existence with thread programming (thread-safety) and memory management [23]. 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), so 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 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.

Nonetheless ten years have passed, just a few stable MPI distributions include a complete support to this version, such as LAM-MPI, Fujitsu, NEC and MPICH2. 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.

**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 [24].

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. In addition, grid environments might provide a computational power in a scale never seen before, so coping with the intention of most of the MPI applications, that of high performance.
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 interoperating 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, as an user interface, it stands on the top of a library that implements this interface accordingly to the specification, independent on the resources. However, the interoperability between different vendor implementations of the standard is not the rule.

Dynamic process creation and management features The MPI-2 standard [23] 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.

Even though 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.

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 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.

As stated before, Grids are inherently hierarchical infrastructures usually composed by a large number of resources. In order to coordinate these resources, collective communication is a strong requirement. In this sense, the MPI standard offer many collective communication modes and primitives, providing an advanced support to the definition of collective communication so that the control over resources can be made in parallel.

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 (section 2.3) and analyzed in more details in the following paragraphs.
**MPI is too much 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. This process should happen not just at beginning of the execution but also during the execution. 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 communicators exist, but it does not offer an adequate support to 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 identify the source and destination through identifiers. This can be a constraint in environments where hosts arrive and leave the infrastructure at any time because one missing message may prevent an application to keep executing. The same happens when a node fails.

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 a static entity itself.

**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, 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 arise.

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

**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, addresses flat environments and do not offer a mechanism to deal with heterogeneity on networks. Besides, MPI presupposes direct all-to-all node access, while it is commonplace grid infrastructures where not all nodes have access to all of the other nodes.

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 other 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 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.

2.3.1 PACX-MPI

The PACX-MPI [25], 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 [26]:

- 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.

2.3.1.1 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. 2.1. 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.
2.3.1.2 The Main Features

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 2.2 shows how the global and local ranks are organized.

Optimization of Global Communication  For the obtention of a better performance on collective operations, two algorithms are used within PACX-MPI [25]:

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

2.3.2 MPICH-G2

The MPICH-G2 [14], 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.

2.3.2.1 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.

**MPICH** MPICH [27] 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.

**Globus** The Globus Toolkit is a collection of software components designed to support the development of applications for high-performance distributed environments, or grids [6]. 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.

2.3.2.2 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. 2.3. 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. 2.3.

![Figure 2.3: MPICH-G2 Architecture](image)

2.3.2.3 The Main Features

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 2.4 shows a scheme of this mechanism.

**Figure 2.4:** 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.

### 2.3.2.4 High Performance Heterogeneous Communication

Different from its previous implementation (MPICH-G [28]), that have all the communication made by the `nexus` library [29], the MPICH-G2 handles itself all the communication directly. According to [14], 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 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.

**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.

2.3.3 MPICH/Madeleine

MPICH/Madelaine [12, 13], 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 2.3.2), 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.

2.3.3.1 Building Blocks

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

**MPICH** As presented in the section 2.3.2.1, 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 2.3.2.1.

**Madeleine** The Madeleine programming interface [12] 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.

2.3.3.2 Architecture

The MPICH/Madeleine architecture is mainly based on the MPICH layered implementation, as described in the section 2.3.3.1. In the MPICH structure, just the lower communication layer (ADI) is re-implemented by MPICH/Madeleine.

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 [30] 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 2.5.

![Figure 2.5: MPICH/Madeleine Architecture](image)

2.3.3.3 Main Features

**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 processes are not capable of using the same protocol.

**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 [13]. This mechanism creates the possibility of communication between nodes that do not have a direct physical link.

This mechanism depends on a manual configuration file defining the topology, 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.

### 2.3.4 H₂O MPI

The H₂O MPI [31], 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 H₂O 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 H₂O 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 H₂O to provide value added features like dynamic staging, updating of proxy modules and selective, streamlined functionality as appropriate to the given situation.

H₂O 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 be treated apart.

The following subsections present the main H₂O MPI building blocks, its architecture and also the main features included on this tool.

#### 2.3.4.1 Building Blocks

The H₂O MPI is built on the top of the FT-MPI [32] and with the support of the H₂O framework on crossing-firewall configurations.

**The FT-MPI** The H₂O MPI [31] 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$_2$O 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.

**The H$_2$O 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 (Common 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 forwards 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.

**2.3.4.2 Architecture**

The architecture presented by the H$_2$O MPI is made of pluglets within specific H$_2$O kernels, located in strategic points of the infrastructure (e.g. frontends) and H$_2$O proxy pluglets in each of the resources involved in the computation. The figure 2.6 depicts these two main elements and their connection in a point-to-point communication between two resources with a firewall between them.

![H$_2$O MPI Architecture: simple example](image)

According to [31], some optimizations related to the communication between pluglets were in progress. These enhancements would guarantee a better performance not just in point-to-point com-
munication but also in collective communication. However, more information about these enhancements were not provided yet.

2.3.4.3 Main Features

**Message Tunneling**  Due to the fact that H$_2$O 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$_2$O 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 daemons).

**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 [31], the overhead can be greatly reduced.
Chapter 3

Context

This chapter presents the context on which this research work is developed. We present the ProActive grid middleware [15] and also the tools that serve as a support to the work developed.

3.1 The ProActive Grid Middleware

The ProActive middleware [15] 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), code mobility, 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 [33].

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

3.2 ProActive Deployment Framework

The ProActive middleware includes a powerful deployment model and framework [15]. 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 use various protocols, rsh, ssh, Globus, LSF, etc., for the creation of the JVMs needed by the application or even MPI applications. 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 Virtual Nodes (VNs), that are potentially composed by many Nodes. The VNs are initially described in XML descriptors and are defined in three different aspects: the mapping of VNs to Nodes and to JVMs, the way to create or to acquire JVMs and the way 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.

3.3 ProActive MPI Code Wrapping

The Code Wrapping Mechanism, recently introduced in the ProActive middleware proposes a simple wrapping method designed to automatically deploy 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 exchange of messages between the two worlds. Furthermore, this feature might add capabilities of communicating between two MPI processes potentially located at different domains under firewalls or even private IPs;

- 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;

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

The Grid Component Model (GCM) [4], 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 [2] 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 specificities 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.
Chapter 4

A Proposal for Grid-Aware Hierarchical Communication

This chapter presents a proposal for grid-aware hierarchical communication. First, we expose the main reasons that lead us to take our approach. After, we present the specification and some pertaining implementation issues and details.

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. So, 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 [34].

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. However, the idea of hierarchical communication does not exists in the standard. Thus, we believe that an extension to the standard would enable it to grid computing and, at same time, facilitate grid platforms to be used in 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 referred in previous sections, we intend to keep as much as possible the MPI programming style. In order to do so, we will provide some extensions to already existing MPI abstractions, like communicators, to include support for hierarchical communication and topology discovery, yet having the grid complexities hidden by the framework.

4.2.1 New MPI communicators

Ideally, we consider a grid (Level 1 of the hierarchy, or just L1 for simplicity) as a five levels architecture:

- the second level (L2) 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;
- the third level (L3) consists of the set of clusters defining a grid node. These clusters are typically interconnected by faster links;
- the fourth level (L4) consists of the set of processing nodes defining a cluster. On each grid node, the aggregated number of processing nodes is between a few hundreds to a thousand nodes, usually with a fast interconnection network;
- the fifth and lowest level is characterized by the architecture of a node: single versus multiple-processor systems, single versus multiple-core systems and memory structure (SMP, NUMA, etc.).

As standard MPI already addresses the fourth level and the operating system the fifth, we introduce two new communicators to deal with the first, second and third levels: (i) the `MPI_COMM_SITE` communicator that contains references to all the allocated nodes within a given site of the grid and (ii) the `MPI_COMM_GRID`, that references all the nodes allocated in the grid.

![Hierarchical Communicators and ranks](image)

**Figure 4.1: Hierarchical Communicators and ranks**

The figure 4.1 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_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;

- 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 table 4.3.3). 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 prototype, we do 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 will 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.

All the communicators, with the exception of the MPI_COMM_CLUSTER, that is an alias to the MPI_COMM_WORLD, are actually implemented at the interface level, i.e. there are no MPI data structures corresponding to them at the level of the standard MPI implementation. So, they cannot be used to perform operations over communicators like MPI_Comm_Split, MPI_Comm_Merge and MPI_Comm_Dup.

### 4.3 Design of the Component-Based Support

#### 4.3.1 Architecture

The support to the primitives presented in the previous section is based on a layered architecture according to the figure 4.2. 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, there is the control whether the communication will go through the native MPI implementation (intra-cluster communication) or through the components and the underlying JVM (inter-cluster communication). And, at the bottom, we have the operating system (OS) that takes care of process management and network.
4.3.2 The underlying component infrastructure

Behind the abstraction of the hierarchical communicators, the infrastructure is formed of multiple concurrent MPI computations and a deployed component framework wrapping these native processes and offering support to collective and inter-cluster communication (through "non-wrapping" additional components). All those components are organized so as to reflect the hierarchical organization of the grid resources allocated to the application. On this infrastructure, all the components receive a 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 4.3 shows how the identifiers of a component hierarchy equivalent to the figure 4.1 are organized.

Two types of components were designed to model the grid infrastructure:

- a **wrapper** component (Figure 4.4.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 in charge of the encapsulation of the MPI messages into objects, that also includes information useful to route messages within the component infrastructure (section 4.3.3). It presents just a server (drawn on the left hand side of a component) **Recv** and a client (drawn on the right hand side of a component) **GoUp** 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 4.4.b) is a generic component capable of clustering lower-level components. In our case, a clustering component at level 3 (L3) groups L4 wrapper components, and the clustering component at level 2 (i.e., at the level of sites in the grid hierarchy) groups L3 clustering components. The same for a component at level 1 (a grid) that groups L2 components (sites). These components are responsible for communications between different clusters and sites of the grid. They present a **Recv** server interface that is responsible for receiving requests from lower-level or higher-level components. The **GoUp** client interface is bound to the upper-level clustering component, that represents the site (resp. the grid) where the cluster (resp. the site) (and so the component that represents it) is located. It is a multicast interface just to enable, if needed, to bypass one level of the hierarchy: e.g. all sites of a grid could be directly bound without passing through the upper component that would represent the grid level. The **GoDown** multicast client interface is bound to all of the nodes (resp. clusters or sites) of the given cluster (resp. site or grid) and is capable to perform both, point-to-point communication with a given L4 (resp. L3 or L2) component, or collective communication, taking profit of the multi-threaded capabilities of the multicast interfaces. Even if this **Recv** interface is not a gathercast interface (as defined in [35]), it can perform gather operations of the received messages within the implementation of the component. Due to its simplicity, it is the approach we have selected for our first implementation.

The figure 4.5 shows the components described above and their bindings, considering the structure of Site 0 of the hierarchy presented in the figure 4.1.

### 4.3.3 Message Routing over the Grid

After launching the environment, a global view of the topology is built and each process creates locally its own view of the topology through a mapping between the communicators and unique identifiers. The following figure shows an example of table (in this case, the table stored in the nodes 0:1:0 and 0:1:1 of the figure 4.3).

By having this information, the decision regarding message routing can be done in the native process. It 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.

#### 4.3.3.1 Point-to-Point Messages

Due to these 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, respectively) are the same, 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. This requires first getting the recipient reference through its GoUp client interface; in the case the source and destination are placed in different sites, this process happens again to an upper-level clustering component, that represents a site in the topology. Having obtained the remote reference of the destination component, it becomes possible to forward directly the message by triggering a ProActive communication between the two remote active objects that wraps 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. First, it may happen that not all pairs of MPI processes are going to communicate. Second, in case MPI processes are relocated on different grid resources at run-time, we have only to update the hierarchy of components accordingly, and propagate the required modifications in the tables stored at the level of wrapper components in order to have up-to-date routing information.

In the case of firewalled clusters or sites, were 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, it requires at least one extra copy for each hop of the path. 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.

4.3.3.2 Collective Messages

The routing of collective communication is slightly more complex than the point-to-point.

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.

For that reason, despite the fact that point-to-point communications can be done directly between wrapper components, in the case of collective communication, 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 an MPI root process 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 L4 to the L3 clustering components and later from them to the L2.

- 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 enables to unblock it.

- Scatter: in the 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 charge 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.

• Barrier: in the case of the barrier primitives, 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.

Besides of the bindings, each clustering component have a queue to store collective messages. The treatment of these messages follows a FIFO (first in, first out) order and this enforces message ordering as done in MPI standard to ensure correctness.
Chapter 5

Evaluation

This chapter presents an 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 point-to-point and collective communications. After, we present some experiments based on classical applications developed/executed using our framework, in comparison with pure MPI versions. Also, this chapter presents a qualitative comparison between the developed framework and already existing tools.

5.1 Experimental Environment

The experimentations were conducted in three clusters of the grid5000 testbed (Sophia-Antipolis, Lyon and Grenoble) and are given by the average of a hundred executions 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 $\text{speedup} = \frac{\text{Time}_{\text{sequential}}}{\text{Time}_{\text{parallel}}}$ and is useful to verify the efficiency obtained out of a parallel application.

The native executions make use of the MPICH 1.2.7p library and the ProActiveMPI ones make use of MPICH v1.2.5, ProActive v3.1 and Java Sun SDK 1.5.6. 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, the number of nodes were divided equally among the clusters, for instance: in the experiments with 60 nodes, the MPI-local-cluster configuration consists in 60 nodes placed within a single cluster, the ProActiveMPI-2-cluster configuration has 30 nodes on each cluster and in the ProActiveMPI-3-clusters 20 nodes on each of the clusters.

5.2 Benchmarks

Point-to-Point Communication  In the first benchmark, we evaluate the point-to-point communication in different scenarios. First, we evaluate the overhead measured in single-site execution. The figure 5.1 shows that despite of the fact that there is a control to decide how to handle messages, the overhead on communication that takes place within a single cluster is negligible.

Differently, in the case of cross-cluster communication 5.2, the usage of components generates a non negligible overhead. The main reasons for this overhead are forwarding of messages through
components and also an increase of latency and decrease of bandwidth due to the shared WAN network.

Even if differences in performance compared with MPI are considerable, we can notice that in the ProActiveMPI version we have just a smooth increase in time even with larger messages. We also emphasize that even if we introduce some overhead, the component framework adds capabilities of inter-cluster communication that would not be possible otherwise.

Due to the existence of a non-dedicated network between the sites, we indicate the standard deviation by the vertical blue bars.

**Collective Communication**  In the second benchmark, we evaluate the performance of the multi-cluster collective communication. This execution regards to the broadcast operation, but the results are extensible to other collective communication operations (gather, scatter and barrier), as they follow the same path in the component hierarchy. In the comparison, we analyze the MPI performance on a single cluster against the ProActiveMPI performance on a multi-cluster grid environment. When using more than one cluster, the total number of nodes was equally distributed throughout the clusters.

According to the results presented in the figure 5.3, it is possible to affirm that we kept the same relation of overhead (about 4x slower), from just a few nodes until 60 nodes spread throughout three clusters. There are two main reason for the differences: first, MPI collective operations are optimized through the creation of a communication tree that improves real (distributed) parallelism, while group calls using component are only multi-threaded. Second, in the current prototype, the collective calls must pass through the components and so, one more hop in the hierarchy must be considered.

Worthy of notice that when the number of cluster increases, so does the performance. The main reason is that the collective interfaces are capable of broadcasting data in a (pseudo) parallel way. Thus, by increasing the number of clusters we also increase parallelism in the component level. Also, we expect this behavior to be kept with more clusters, but this depends on more tests.

**5.3 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 $\pi$ (3.14159265...), representing the embarrassingly parallel master-worker applications; (2) a hierarchical parallel merge-
sort, representing the Divide-and-Conquer approach to build parallel applications; and (3) the Poisson3D solver representing non-embarrassingly application based in data-domain decomposition.  

The idea is that we could expect a similar behavior in applications that follows the same approach. The following section present, in more details, each of the applications and how they were developed. After, we analyze the obtained results.

5.3.1 Applications

(1) Monte-Carlo Simulation  Monte Carlo methods are a widely used class of computational algorithms for simulating the behavior of various physical and mathematical systems. A Monte Carlo algorithm is often used in mathematical problems, which may have many variables, that cannot easily be solved, for example, by integral calculus, or other numerical methods. 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 $\pi$ ($3.14159265...$). It consists on the generation of pseudo-random numbers between 0 and 1 representing a given point on the figure 5.4 and the computation of $\pi$ is based in the verification whether the points are inside or outside of the circle.

Figure 5.4: Pi computation Scheme

---

1The tests with the Poisson3D application are in progress and experimental results should appear in the final version of the report
The area of a quarter of circle is done by:

\[ A = \frac{1}{4} \pi r^2 \]  

(5.1)

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

\[ D = (x_2 - x_1)^2 + (y_2 - y_1)^2. \]  

(5.2)

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 \( \Pi \) as being:

\[ \Pi = \frac{r^2 \cdot \frac{P_{in}}{P_{out}}}{\frac{1}{4} \cdot r^2} = 4 \cdot \frac{P_{in}}{P_{out}} \]  

(5.3)

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. In order to reduce inter-cluster communication, the results are first gathered in the first worker of each cluster before they are sent to the master.

(2) Hierarchical Mergesort  
Mergesort is a well-known sorting algorithm developed by John von Neumann [36] 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 sub-lists. 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 5.5 summarizes these steps for a small array.

![Mergesort example](image)

Figure 5.5: Mergesort example

Due to its divide-and-conquer nature, mergesort can be easily parallelized. Two different versions of the mergesort were developed:
• (i) flat version: this version follows strictly the master-worker pattern 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. The figure 5.6 shows the communication pattern.

• (ii) hierarchical version: this version implements the mergesort by using a hierarchical approach. Initially, the array of numbers is divided in \( n \) sub-arrays, being \( n \) the numbers of clusters of the ProActiveMPI version (2 in our executions). After, this sub-arrays are divided again, recursively within each cluster. The main benefit of the hierarchical version is that the merge process can happen in parallel in each of the clusters. In the case of the ProActiveMPI version, we take care to send one sub-array to each cluster. The figure hier-merge depicts the hierarchical algorithm.

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 Poisson algorithms does not have a master-worker organization. This means that there is no 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 handles neighbor pieces of the matrix.

Our Poisson implementation uses an iterative Jacobi method. It consist in performing several operations, namely gather neighbor’s data and, for each point of the domain we iterate through. In our case, a 3 dimension matrix spread across several computation nodes is used.

The core of the distributed Poisson equation resolution method involves the following steps:

1. Exchange local matrix’ borders with neighbors.
2. compute Jacobi over the local 3 dimension matrix using received borders.
3. compute a residue.
4. send the residue value from all processes to all processes (all-reduce).
5. compute next iteration local matrix.
6. iterate until a certain threshold is reached
Two versions using this algorithm were developed: a pure MPI one and a ProActive-MPI one. They only differ from each other in the communication layer used.

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. In order to reduce inter-cluster communication the matrix is partitioned in slices instead of a mesh (see Figure 5.8). Thus, with a correct sub-matrix distribution, we ensure at most two processes per cluster will need to perform such communication.

![Figure 5.8: Poisson sliced data partition](image)

5.3.2 Results

In the Figure 5.9, we can notice that all the time curves have almost the same shape and that they are 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 behaviour is highly dependent of the design of the application and the amount of communication done.

The difference in execution time between the executions in 2 or 3 clusters are probably due to larger latencies in the communication with the third cluster, that is geographically far 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 3 even. 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 5.10, we can notice that our framework can be considered scalable. Some empirical experiments have shown that, for the monte carlo simulation, this behaviour is stills valid even for a few hundred nodes and that the degradation in performance in these cases is comparable to that of the pure MPI version of the application. Two reasons also helped to keep a good overall performance: first, the monte carlo application does not communicate a large amount of data and second that the monte-carlo was designed to avoid unnecessary inter-cluster communication.

Unlike the monte carlo application, the experiments conducted with the mergesort application presented a bigger degradation in performance when using the component model. 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 at the same time and that this aggregation of resources would not be possible otherwise.

When comparing the graphics 5.11 and 5.12 we can notice the importance of taking into account the topology when designing the parallel algorithm. In the flat version, there are some configurations where the ProActiveMPI version takes about 8 times more than the pure MPI version, against 2.5 times in the worst case of the hierarchical version. This happens because, in the flat version, the communication process happens in a WAN context (so with a greater latency and smaller bandwidth) and we need \( n \) messages to spread the data throughout \( n \) nodes 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 clusters.

The figure 5.13 shows more clearly a comparison between the two versions. From this comparison, we can see that the flat version just have overall computing time reduced until 8 nodes, and in the hierarchical version, we could have an improvement on the performance up to 64 nodes or even more.

From the figures 5.14 and 5.15, we can see that the mergesort algorithm does not scale well with a larger number of nodes. 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 costly as the number of workers increases. The hierarchical approach has a speedup a little better because the merging process happens in parallel, whilst all the merge process happens locally in the manager in the case of the flat version.

Even though for the mergesort we could not keep the good speedup of monte-carlo (figure 5.10), it does not means that the developed prototype does not scales well. A proof of that is that even the pure MPI, that is considered highly scalable, did not obtained a good speedup. Some other 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.

5.4 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 lot 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 (section 2.3).

The table 5.4 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 plataforms 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.

<table>
<thead>
<tr>
<th>Features</th>
<th>Tools</th>
<th>PACK-MPI</th>
<th>MPICH-G2</th>
<th>MPICH-Mad</th>
<th>H2O</th>
<th>ProActiveMPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Support to network/grid protocols/middleware</td>
<td>network protocols</td>
<td>Globus only</td>
<td>network protocols</td>
<td>messaging protocols</td>
<td>network protocols (through MPI) and grid middlewares</td>
<td></td>
</tr>
<tr>
<td>2. Resources Management</td>
<td>no</td>
<td>yes (through Globus GRAM/DUROC)</td>
<td>no</td>
<td>no</td>
<td>yes (PBS, LSF, OAR, Globus, gLite, ...)</td>
<td></td>
</tr>
<tr>
<td>3. Data Management</td>
<td>no</td>
<td>yes (through Globus sandboxing and GlobusGE)</td>
<td>no</td>
<td>no</td>
<td>ProActive File-Transfer</td>
<td></td>
</tr>
<tr>
<td>4. Forwarding/Tunneling of Messages</td>
<td>Forwarding/Tunneling</td>
<td>no</td>
<td>Forwarding/Tunneling (configured by hand)</td>
<td>Forwarding/Tunneling (configured by hand)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Topology-aware Communication</td>
<td>no</td>
<td>yes (implicit)</td>
<td>no</td>
<td>no</td>
<td>yes (explicit and implicit)</td>
<td></td>
</tr>
<tr>
<td>6. API</td>
<td>no</td>
<td>yes (new MPI attributes)</td>
<td>no</td>
<td>no</td>
<td>yes (new MPI CommS and primitives)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the developed prototype with related tools

From this the comparison, 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.
Chapter 6

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 for clusters and the existence of many legacy applications and libraries, the idea of using MPI in grids has 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, we understand that parallel algorithms must be adapted to reflect grid topology in order to obtain a good performance. However, we identified a lack of mechanisms and abstractions to design and develop grid-aware hierarchical MPI applications. Also, we consider 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.

So, we propose 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 native code wrapping within active objects 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. Besides, the extended interface may offer users the adequate abstractions to design parallel algorithms in a hierarchical way addressing grid environments.

Even if we have developed a first simplified prototype, the design was proved to be scalable and, with the necessary improvements, we can expect a performance very close to native MPI implementations. The main expected improvements include:

- Reduce intermediary copies in inter-cluster communication, by replacing the Java-C communication with a library that enables direct access from Java applications to memory allocated outside of the JVM, through the use of the java.nio package.

- An improved component model that avoid, whenever possible, unnecessary bindings and bottlenecks by means of direct bindings for collective communication between components instead of communication forwarded through high-level components.

As a continuation of the work we expect to compare, in performance, our prototype against related tools and also develop a real size application to explore in deep the framework capabilities. The
research subject developed during the stage will also lead to a PhD thesis that will take place in the context of the *DiscoGrid* project [37], that is a 3 years ANR (*Agence Nationale de la Recherche*) funded project. In summary, the research topic of the PhD is the development of a component model and associated API to support MxN communication in the context of physical/numerical high performance simulations. So, this can be understood as a generalization of grid programming with our hierarchical MPI extension, y introducing the concept of hierarchical addressable process that encompasses more than one MPI process.

This work also create opportunities to other related research topics. For instance:

- Support of MPI2 features of dynamic creation of process and management, adding to MPI the support to dynamicity in resources acquisition/release;

- Support to fault-tolerance in grid-aware applications, by means of checkpoint-restarting MPI applications and dynamic reconfiguration of bindings between components;
Bibliography


