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

MÁRCIA CRISTINA CERA

Providing Adaptability to MPI Applications on Current Parallel Architectures

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

Dr. Philippe O. A. Navaux
Advisor

Dr. Nicolas Maillard
Coadvisor

Dr. Olivier Richard
Partial Doctoral Fellowship Advisor

Porto Alegre, August 2012

CIP – CATALOGING-IN-PUBLICATION

Márcia Cristina Cera,

Providing Adaptability to MPI Applications on Current Parallel Architectures /

Márcia Cristina Cera. – Porto Alegre: PPGC da UFRGS, 2012.

150 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2012. Advisor: Philippe O. A. Navaux; Coadvisor: Nicolas Maillard; Partial Doctoral Fellowship Advisor: Olivier Richard.

1. MPI, Adaptability, Malleability, Explicit Task Parallelism. I. Navaux, Philippe O. A.. II. Maillard, Nicolas. III.. Richard, Olivier. IV. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Herdeiro da Pampa Pobre

Composição: Gaúcho da Fronteira - Vaine Darde

Mas que pampa é essa que eu recebo agora
Com a missão de cultivar raízes
Se dessa pampa que me fala a estória
Não me deixaram nem sequer matizes?

Passam as mãos da minha geração
Heranças feitas de fortunas rotas
Campos desertos que não geram pão
Onde a ganância anda de rédeas soltas

Se for preciso, eu volto a ser caudilho
Por essa pampa que ficou pra trás
Porque eu não quero deixar pro meu filho
A pampa pobre que herdei de meu pai

Herdei um campo onde o patrão é rei
Tendo poderes sobre o pão e as águas
Onde esquecido vive o peão sem leis
De pés descalços cabresteando mágoas

O que hoje herdo da minha grei chirua
É um desafio que a minha idade afronta
Pois me deixaram com a guaiaca nua
Pra pagar uma porção de contas

ACKNOWLEDGEMENTS

I would like to thank all those contributed in the development of this thesis and to do that I must write some words in Portuguese.

Preciso dizer mais uma vez um muito obrigado aos meus orientadores Prof. Philippe O. A. Navaux e Prof. Nicolas Maillard. Além de exemplos profissionais vocês foram mais que mestres, foram amigos. Obrigada por acreditarem em mim.

Agradeço também ao Prof. Olivier Richard, pela acolhida e ensinamentos durante minha estada na França. (Je remercie également au Prof. Olivier Richard, pour l'accueil et l'enseignement au cours de mon séjour en France.)

Agradeço à Universidade Federal do Rio Grande do Sul (UFRGS), em especial ao Programa de Pós-Graduação em Computação (PPGC), e a todos os professores que contribuíram para a minha formação. Adicionalmente, agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo suporte financeiro durante o doutorado e doutorado sanduíche.

Agradeço a Banca Examinadora, Andrea Charão, Jairo Panetta, Alexandre Carissimi, pelas preciosas considerações sobre o meu trabalho.

Aos meus colegas de GPPD (Grupo de Processamento Paralelo e Distribuído) agradecerei eternamente pelas boas discussões, pelo amadurecimento científico, pelas ajudas na correção do inglês desta tese e também pelos momentos de descontração.

A todos os meus amigos também os agradeço pelo incentivo.

Por último mas não menos importante, agradeço a minha família pelo apoio incondicional e peço desculpas pelas ausências.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

AJS	Adaptive Job Scheduler
AMPI	Adaptive MPI
API	Application Programming Interface
CCS	Converse Client Server
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
D&C	Divide and Conquer
FJT	Folding by JobType
HPC	High-Performance Computing
KAAPI	Kernel for Adaptative, Asynchronous Parallel and Interactive programming
LRU	Least Recently Used
MPI	Message-Passing Interface
MPMD	Multiple Programs, Multiple Data
OpenMP	Open Multi-Processing
OS	Operating System
PCM	Process Checkpointing and Migration
PCMD	PCM Daemon
PE	Processing Element
QoS	Quality of Service
RMS	Resource Management System
SPMD	Single Program, Multiple Data
TBB	Threading Building Blocks
UE	Units of Execution

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Currently, adaptability is a desired feature in parallel applications. For instance, the increasingly number of user competing for resources of the parallel architectures causes dynamic changes in the set of available processors. Adaptive applications are able to execute using a set of volatile processors, providing better resource utilization. This adaptive behavior is known as malleability. Another example comes from the constant evolution of the multi-core architectures, which increases the number of cores to each new generation of chips. Adaptability is the key to allow parallel programs portability from one multi-core machine to another. Thus, parallel programs can adapt the unfolding of the parallelism to the specific degree of parallelism of the target architecture. This adaptive behavior can be seen as a particular case of evolutivity. In this sense, this thesis is focused on: *(i)* malleability to adapt the execution of parallel applications as changes in processors availability; and *(ii)* evolutivity to adapt the unfolding of the parallelism at runtime as the architecture and input data properties. Thus, the open issue is *"How to provide and support adaptive applications?"*. This thesis aims to answer this question taking into account the MPI (Message-Passing Interface), which is the standard parallel API for HPC in distributed-memory environments. Our work is based on MPI-2 features that allow spawning processes at runtime, adding some flexibility to the MPI applications. Malleable MPI applications use dynamic process creation to expand themselves in growth action (to use further processors). The shrinkage actions (to release processors) end the execution of the MPI processes on the required processors in such a way that the application's data are preserved. Notice that malleable applications require a runtime environment support to execute, once they must be notified about the processors availability. Evolving MPI applications follow the explicit task parallelism paradigm to allow their runtime adaptation. Thus, dynamic process creation is used to unfold the parallelism, *i.e.*, to create new MPI tasks on demand. To provide these applications we defined the abstract MPI tasks, implemented the synchronization among these tasks through message exchanges, and proposed an approach to adjust MPI tasks granularity aiming at efficiency in distributed-memory environments. Experimental results validated our hypothesis that adaptive applications can be provided using the MPI-2 features. Additionally, this thesis identifies the requirements to support these applications in cluster environments. Thus, malleable MPI applications were able to improve the cluster utilization; and the explicit task ones were able to adapt the unfolding of the parallelism to the target architecture, showing that this programming paradigm can be efficient also in distributed-memory contexts.

Keywords: MPI, Adaptability, Malleability, Explicit Task Parallelism.

Provendo Adaptabilidade em Aplicações MPI nas Arquiteturas Paralelas Atuais

RESUMO

Atualmente, adaptabilidade é uma característica desejada em aplicações paralelas. Por exemplo, o crescente número de usuários competindo por recursos em arquiteturas paralelas gera mudanças constantes no conjunto de processadores disponíveis. Aplicações adaptativas são capazes de executar usando um conjunto volátil de processadores, oferecendo uma melhor utilização dos recursos. Este comportamento adaptativo é conhecido como maleabilidade. Outro exemplo vem da constante evolução das arquiteturas multi-core, as quais aumentam o número de cores em seus chips a cada nova geração. Adaptabilidade é a chave para permitir que os programas paralelos sejam portáveis de uma máquina a outra. Assim, os programas paralelos são capazes de adaptar a extração do paralelismo de acordo com o grau de paralelismo específico da arquitetura alvo. Este comportamento pode ser visto como um caso particular de evolutividade. Nesse sentido, esta tese está focada em: *(i)* maleabilidade para adaptar a execução das aplicações paralelas às mudanças na disponibilidade dos processadores; e *(ii)* evolutividade para adaptar a extração do paralelismo de acordo com propriedades da arquitetura e dos dados de entrada. Portanto, a questão remanescente é *"Como prover e suportar aplicações adaptativas?"*. Esta tese visa responder tal questão com base no MPI (*Message-Passing Interface*), o qual é a API paralela padrão para HPC em ambientes distribuídos. Nosso trabalho baseia-se nas características do MPI-2 que permitem criar processos em tempo de execução, dando alguma flexibilidade às aplicações MPI. Aplicações MPI maleáveis usam a criação dinâmica de processos para expandir-se nas ações de crescimento (para usar processadores extras). As ações de diminuição (para liberar processadores) finalizam os processos MPI que executam nos processadores requeridos, preservando os dados da aplicação. Note que as aplicações maleáveis requerem suporte do ambiente de execução, uma vez que precisam ser notificadas sobre a disponibilidade dos processadores. Aplicações MPI evolutivas seguem o paradigma do paralelismo de tarefas explícitas para permitir adaptação em tempo de execução. Assim, a criação dinâmica de processos é usada para extrair o paralelismo, ou seja, para criar novas tarefas MPI sob demanda. Para prover tais aplicações nós definimos tarefas MPI abstratas, implementamos a sincronização entre elas através da troca de mensagens, e propusemos uma abordagem para ajustar a granularidade das tarefas MPI, visando eficiência em ambientes distribuídos. Os resultados experimentais validaram nossa hipótese de que aplicações adaptativas podem ser providas usando características do MPI-2. Adicionalmente, esta tese identificou os requisitos no nível do ambiente de execução para suportá-las em clusters. Portanto, as aplicações MPI maleáveis melhoraram a utilização de recursos de clusters; e as aplicações de tarefas explícitas adaptaram a extração do paralelismo de acordo com a arquitetura alvo, mostrando que este paradigma também é eficiente em ambientes distribuídos.

Palavras-chave: MPI; Adaptabilidade; Maleabilidade; Tarefas Explícitas.

1 INTRODUCTION

1.1 Motivation

Nowadays, the challenge in the development of parallel programs is to provide adaptability. Users of current parallel architectures compete for resources such as processors, bandwidth, memory, and so on. This competition happens even on large-scale environments, *i.e.*, to scale the parallel environment does not guarantee to serve the users' demand. A main consequence of this competition is a dynamic availability of the resources. In other words, the amount of resources allocated to a user (or application) may change at runtime according to the environment utilization. For example, consider that an application is able to run on unused processors of a cluster aiming its full utilization. At starting time, the application uses the whole set of unused processors, and this set will be updated at runtime as other applications request or release processors. To allow the full cluster utilization, the application must be able to adapt itself or, in other words, it must be malleable.

On the other hand, many researches are focused on other adaptive applications, which can adapt their execution to a specific degree of parallelism as the target architecture (REINDERS, 2007; LEISERSON, 2009; AYGUADÉ et al., 2009). Today, multi-core architectures are widely spread, and their number of cores is increasing from one chip generation to another. Thus, studies aim to provide means to explore the algorithm parallelism at runtime, without considering how many cores exits on the target architecture. In this sense, the explicit task parallelism programming paradigm allows the definition of algorithmic abstract tasks and their dependencies. Thus, the parallelism can be unfolded during the application execution, guiding the explicit creation of tasks on demand. Notice that the abstract tasks are defined on algorithmic level, which means that the potential parallelism to be explored depends on the size of the input data.

To achieve efficiency on these scenarios, the key feature desired from parallel applications is to be adaptive. CUNG et al. (2006) define adaptive algorithms as those able to consider the availability of the resources or the input data properties to guide their runtime adaptive actions. According to FEITELSON; RUDOLPH (1996) there are two class of parallel jobs that include some flexibility: *malleable jobs* that are able to adapt themselves to changes in the number of processors at runtime; and *evolving jobs* that may require unpredictable changes in the number of required processors at runtime due to inherent irregularity of the application (more details in Section 2.2). As evolving applications are hard to be supported, many initiatives try to schedule the irregular worload into the available processors. In this sense, the explicit task parallelism can be seen as one these initiatives. Furthermore,

GHAFOOR (2007) defines adaptive applications as those have malleable or evolving behaviors according to Feitelson's classification. This thesis is focused on applications with both levels of adaptability: malleability to adapt to volatile processors and evolving to adapt to unpredictable needs.

In this context, there is a critical open issue: *"How to provide and support adaptive applications on current parallel architectures?"*. The answer of this question rises in three fronts: programming level or how to use current programming techniques, mechanisms, and tools to develop parallel applications able to perform the required reactions at runtime; environment level or how to provide communications between applications and the runtime environment to satisfy the adaptability requirements; and application scheduling level or how to provide on-line scheduling of applications workload to ensure performance under adaptive conditions. This thesis aims to investigate these aspects to MPI (Message-Passing Interface) (GROPP; LUSK; SKJELLUM, 1994) applications.

MPI is the focus of our work because it is widely used in HPC (High-Performance Computing) applications to distributed-memory environments. Furthermore, MPI-2 specification defines interfaces to include some flexibility into MPI applications such as to spawn new MPI processes at application runtime and establish client-server communications among processes. Our goal is to take advantage of this flexibility to implement MPI applications able to adapt themselves at runtime. For instance, to get malleability in MPI applications, it is required means to increase and decrease the number of used processors at runtime. MPI-2 dynamic process creation can help to provide this behavior since it allows to change the number of processes at runtime (more details in Section 4.2). To have evolving MPI applications, dynamic process creation can also help to unfold the parallelism by the creation of new MPI tasks (MPI processes) at runtime (more details in Section 4.3).

However, although dynamic process creation helps to provide adaptability, it brings some challenges to provide and support adaptive MPI applications. Here, to provide means to offer the programming issues to develop these applications, and to support means to offer the runtime issues to allow the execution of the applications in current architectures. For example, once processes are spawned during the application execution, their physical location, *i.e.* their mapping, must be defined at runtime. Dynamic processes perform part of the applications workload, but the granularity of these processes must be adjusted at runtime to ensure performance. Furthermore, adaptive MPI applications only can be executed if the runtime environment is able to support their adaptive actions, *i.e.* their dynamicity. Additionally, this support requires interactions between adaptive MPI applications and the runtime environment. This thesis investigates these issues and proposes solutions aiming to provide and support adaptive MPI applications in current parallel architectures.

1.2 Hypothesis

Given the current tendency on parallel programming, this work suggests the hypotheses that the MPI-2 features can provide the development of adaptive MPI applications. In addition, providing the required runtime environment support, these applications can be executed on current parallel architectures.

1.3 Objectives

The main goal of this thesis is to provide and support Adaptive MPI applications in current parallel architectures.

In this sense, our specific goals are:

- To provide adaptive MPI applications using MPI-2 features. Adaptive applications aimed are:
  - Malleable MPI Applications able to adapt themselves to changes in the number of available processors at runtime;
  - Evolving MPI Applications able to adapt the application execution as the problem irregularity through the unfolding of the parallelism at runtime.
- To investigate the challenges and issues to provide and support adaptive MPI applications, proposing solutions as the current features of parallel architectures. This thesis investigates mainly:
  - The mapping of dynamically spawned MPI processes;
  - The adjusting of MPI tasks granularity aiming to balance the applications workload at runtime;
  - Ways to support the interactions between MPI applications and runtime environment to allow adaptive actions.
- To analyse the performance of adaptive MPI applications on clusters of computers with multi-core nodes.

To achieve these goals, we organized the remaining of this text as described below.

1.4 Outline

This thesis is structured in two parts: **Part I** presents background and related works about adaptability; and **Part II** explains how to provide adaptability into MPI applications. In the first part, Chapter 2 contextualizes the needs of adaptability in current parallel architectures, the requirements to support it, and the program structures able to deal with such a kind of scenarios. Additionally, Chapter 3 shows how to the related works deal with runtime issues of adaptive applications. In the second part of this thesis, Chapter 4 answers the question: *"How to provide adaptability using MPI?"* taking into account the MPI-2 features. To allow the execution of adaptive MPI applications, Chapter 5 describes how to run malleable MPI applications in clusters of computers, while Chapter 6 shows how to provide evolving MPI applications through the explicit task parallelism. Finally, Chapter 7 concludes this thesis with our final remarks, contributions and future works.

Part I

Adaptability: Background and Related Works

2 CONTEXT: ADAPTABILITY IN PARALLEL ENVIRONMENTS

Adaptability is the key to allow an efficient utilization of the current parallel architectures. The goal of this chapter is to present the context that requires adaptability in these parallel environments. In a first moment, the nomenclature adopted in this thesis is presented in Section 2.1. Then, there are the main concepts related to adaptability reached in literature – Section 2.2. In the following, Section 2.3 describes some parallel scenarios aiming to show the main features and challenges in this kind of environment. Afterward, Section 2.4 gathers the requirements to support adaptability in the parallel architectures as well as in the applications. Section 2.5 shows how to supply the adaptability requirements in the parallel program structures, and their adoption in the related works. Finally, Section 2.6 concludes this chapter highlighting the main issues exposed in this chapter.

2.1 Adopted Taxonomy

The goal of this section is to define the taxonomy that will be used in this thesis. Some examples are used to clarify the nomenclature and they will appear in the remaining of the text. Figure 2.1 shows the correlation among the nomenclature. We divided in two parts: the first exposes the programming nomenclature (Section 2.1.1); and the second shows the runtime environment nomenclature (Section 2.1.2).

2.1.1 Programming Nomenclature

Programming model: represents an abstraction of computer systems allowing the development of programs taking into account architectural features. For instance, a programming model defines the way in which processors are interconnected: shared-memory - processors share a common memory region and the information exchanging happen through concurrent access to shared data structures; distributed-memory - each processor has its own memory region and the information must be transfered from one to another using a network interface; and hybrid - processors access their own memory region as well as a shared one;

Programming paradigm: represents the style or the methodology employed in the development of programs. In other words, a programming paradigm defines how to the basic units of the programs are represented. For instance, object-oriented programming is a paradigm based on objects to represent

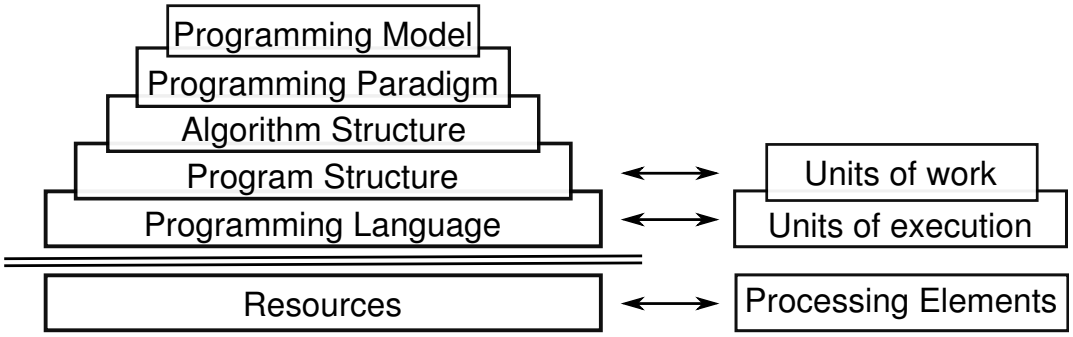


Figure 2.1: Nomenclature adoted in our work.

the programs data structures. Furthermore, the programming paradigm can be linked with issues defined by the programming model. For example, the message-passing programming paradigm is related to the distributed-memory programming model, as well as the multi-threading paradigm with shared-memory one;

Algorithm structure: represents the algorithm organization used to become easy to solve a target problem. For instance, there are problems that can be seen as a set of linear tasks, in which the Task Parallelism algorithm structure can efficiently solve them. In addition, when these set of tasks perform recursive decomposition, instead of Task Parallelism, it is more convenient to use Divide and Conquer algorithm structure. These examples are based on tasks, but the algorithm structures can also be based on data decomposition or flow of data;

Program design or structure: represents the conversion of the algorithm structures into source codes. In other words, a program design represents the programming structures used to support or provide the behavior expected into the algorithmic level. For instance, the set of tasks of the Task Parallelism algorithm structure can be implemented as Master/Worker or SPMD (Single Program, Multiple Data) program structures, while Divide and Conquer algorithms are implemented using a Fork/Join program structures;

Programming Language: gathers the means to compose source codes, offering implementations of the programming structures as the program design. For instance, the Master/Worker program design requires units of execution with two different behaviors: master and worker. The programming language offers means to create units of execution and implement them with different behaviors. Furthermore, a programming language can support more than one programming paradigm and allows handling the abstractions defined by the programming models. For example, the C language is procedural by default, but it can implement the message-passing and multi-threading programming paradigm helped by specific libraries.

2.1.2 Runtime Environment Nomenclature

Resources: in parallel environments aimed in this thesis, resources mean the processors, memory, bandwidth, storage, etc., that are available to the users. Our focus was in issues about processors availability;

Units of Execution (UE): represent the mechanism offered by the programming language to perform the operations and actions defined by the program structure. In the context of our work, the units of execution (UEs) aimed are processes and threads. This terminology is based on MATTSON; SANDERS; MASSINGILL (2004);

Units of Work: represent the workload that must be performed by the units of execution. Some authors name units of work as tasks, but to avoid misunderstood with the definition of task in MPI and in explicit task parallelism, we opted by units of work;

Processing Elements: represent the computational hardware on which the units of execution perform. This thesis is focused on processors and cores as processing elements (PEs). This terminology is also based on MATTSON; SANDERS; MASSINGILL (2004).

2.2 Contextualizing Adaptability

Nowadays, large-scale systems often have a dynamic or volatile set of processors, which become available and are released during the application execution[1]. Furthermore, multi-core architectures are also figuring in these large-scale architectures, which requires ways to profit of their high degree of parallelism. These issues can be addressed using adaptability to allow the application reaction to changes in processors availability, as well as to adjust the application behavior to specific degrees of parallelism. According to CUNG et al. (2006), adaptive algorithms are able to use strategic decisions based on the availability of the resources or on the input data properties, both discovered at runtime. Notice that the size of the input data is related to how much parallel an application can be.

However, the runtime environment must support the adaptive behavior of the applications. This support can mean the spread of the dynamic availability of the processors or the spread of the desired degree of parallelism. Part of this support comes from the Resource Management Systems (RMS). Applications are named as jobs in the RMS context, and it is responsible for schedule and to ensure jobs execution. In general, RMS decisions are based on scheduling policies, which aim to maximize the utilization of the resources and the throughput of the jobs (to serve more users), and reduce the average response time, among other goals. The support of adaptability in RMS comes from improvements in the scheduling decisions, in such a way that issues about the dynamic availability and the degree of parallelism can be used on on-the-fly decisions.

In the following, we describe a popular parallel job classification according to the RMS view, aiming to show the complexity behind the support of adaptability.

2.2.1 Classification of the Parallel Applications: RMS View

FEITELSON; RUDOLPH (1996) proposed a job classification based on who decides the number of processors (user or RMS) and when such a decision is taken (at starting or at runtime). As shown in Table 2.1, Feitelson and Rudolph classifies parallel jobs as:

[1] In this document, we name processors with a dynamic availability as **volatile**.

Table 2.1: Parallel jobs classification according to Feitelson and Rudolph.

Who decides	**When is it decided**	
	At starting time	*At runtime*
User	Rigid	Evolving
RMS	Moldable	Malleable

- **Rigid jobs:** require a certain number of processors according to a user decision. All processors must be provided at starting time and do not soffer any change during the job execution;
- **Moldable jobs:** also performs on a fixed number of processors defined at starting time, but RMS decides how many ones. To guide RMS decisions, the users provide a range (minimum and maximum number) of processors upon which the application can perform efficiently. At starting time, the application configures itself as the number of allocated processors;
- **Malleable jobs:** may adapt themselves to changes in the number of available processors. The changes are required by RMS during the jobs execution, and these jobs must have some flexibility to react properly to changes at runtime. In practice, malleable jobs also have a range of processors representing the application requirements (*i.e.*, the minimum and maximum number of processors in which the parallel application can perform efficiently), which must be known by RMS;
- **Evolving jobs:** may require changes in the number of available processors due to unpredictable variations on application workload. These variations change the applications needs at runtime, which must be supplied by RMS to allow completing their execution. In practice, evolving jobs have an initial need, which is informed and supplied by RMS at starting time. During the execution, the application need changes without any previous knowledge, and RMS must be able to supply it.

Regarding the Feitelson and Rudolph classification, there are two classes of jobs that include some flexibility: malleable – adaptation to volatile processors; and evolving – adaptation to changes in the application workload. In this sense, GHAFOOR (2007) defines adaptive applications, according to the Feitelson and Rudolph classification, as malleable or evolving applications. Ghafoor's work models and simulates an adaptive parallel system in a distributed-memory environment. Although his adaptive applications include evolving ones, Ghafoor only considers malleable applications in his study. The reason is the difficulty to model an RMS system able to supply the unpredictable needs of evolving applications.

To support evolving applications is a challenge. Aiming to solve it, some initiatives proposed solutions without involve the RMS, trying to solve imbalances caused by the workload variations on application side. These initiatives are introduced in the following.

2.2.2 Dealing with Unpredictable Needs

The challenge of supporting evolving applications is that RMS cannot ensure that it will be able to supply all unpredictable requests of changes in the application set of processors. This can be easily understood looking to applications composed by multiple phases: each phase has a proper degree of parallelism and, thus, requires different amounts of processors. For instance, suppose that an evolving application starts using 10 processors and after some time it requires more 10 processors, thanks to an unpredictable workload increase. If there are enough unused processors to attend this extra needs, RMS can allocate them to the application. However, many situations can be imagined in which the extra needs cannot be attended, for example, if the number of unused processors is not enough (in the case, less than 10), if there is no unused processor, if the evolving application already uses all parallel environment processors, and so on.

To despite these hardly issues, many recent studies provide solutions to workload imbalances on the application side (REINDERS, 2007; AYGUADÉ et al., 2009; LEISERSON, 2009). They explore the concept of explicit task parallelism, which is a simple and elegant programming paradigm that allows unfolding parallelism of the algorithms at runtime. In other words, unfold the parallelism means to adapt the application execution to the specific degree of parallelism of the architectures through the explicit creation of abstract tasks on demand. In the development of explicit task applications, the programmer identifies independent units of work (abstract tasks), the dependencies among them, and the runtime scheduler is responsible to balance the workload distribution (MATTSON; SANDERS; MASSINGILL, 2004).

A key issue for explicit task parallelism is to provide load balancing. Moreover, decisions must be taken at runtime, according to the aimed degree of parallelism and the size of input data. The most used strategy to provide on-line scheduling is the Work Stealing. In few words, this strategy considers that each element in the computation[2] has a queue of ready tasks to be consumed when necessary. If a queue become empty, the work stealing starts: it (the thief) chooses randomly another element (the victim) to steal some ready tasks (BLUMOFE et al., 1996). If the queue of the victim is also empty, the thief repeats the procedure choosing another element, until eventually steal some tasks (the strategy includes a procedure to avoid that the stealing requests are demanded indefinitely). This on-line algorithm is theoretically proved efficient to fully strict programs (well-structured computations) in terms of time, space, and communication (BLUMOFE et al., 1996).

Once we presented the background about adaptability in current parallel environments, the next section will illustrate some practical scenarios and the impact of their demands.

2.3 Typical Parallel Scenarios of Adaptability

Large-scale environments often have volatile resources due to users' competition for processors, bandwidth, memory, and so on. Dealing with volatile resources as a whole is complex, because it involves issues such as to monitor their usage, to take decisions and to react at runtime. Some researches show that on-the-fly reconfigu-

[2]In some implementations of Work Stealing strategy, each processor has a queue of tasks. In others, each thread or process has a queue of tasks. For a general introduction of the strategy, we opt to have elements of the computation with queues of tasks.

rations are able to improve the application performance (DU et al., 2004; BOERES et al., 2005; WEATHERLY et al., 2006; SUDARSAN; RIBBENS; FARKAS, 2009). They use rules or policies to take reconfiguration decisions according to the load information collected by monitoring applications and resources. These initiatives are designed taking into account specific features of controlled environments, and thus, cannot be easily adopted in a generic context. In general, they hardly scale since require that the collecting, storing and processing the load information, also scale with the system.

To avoid issues as described above, many researches considered only the dynamicity in the level of processors. Thus, the processors have only two states possible: available or not. So, if one processor is available, this means that it is offering 100% of its power to compute the application. In addition, this is very closer to the exclusive allocation of the processors implemented by many RMS systems: when processors are allocated to an application, they are exclusively used by it. However, this environment is improved to allow the applications receive new or lose some processors at runtime, as the RMS decisions. According to definitions presented in Section 2.2, these applications are named malleable (FEITELSON; RUDOLPH, 1996; BUISSON, 2006; GHAFOOR, 2007), *i.e.* they have a constant computational workload and are able to adapt to changes in the number of processors at runtime.

Aiming to clarify the situations that can change the processors availability in a practical scenario, there are descriptions of some use cases in the following.

2.3.1 Use Cases with Volatile Processors

To illustrate the use cases, we will consider a typical scenario in which an RMS receives submissions of parallel applications from users. During the submission, the users provide to RMS all required information to execute the applications, such as the name of binary, the number of processors, the execution time estimate, the path of input files, and so on. RMS allocates processors to applications following a scheduling policy that takes into account the overall number of processors, as well as the sets of running and the waiting applications. When there are fewer processors than those required by an application, it is inserted in a pending queue until enough processors become available to serve it. In this context, we can identify two situations that may cause changes in the availability of the processors:

Submission of new applications: When a new application is submitted, RMS tries to allocate the required number of processors; otherwise the application goes to the pending queue. However, if RMS is able to handle malleable applications, it may analyse if there are some able to release enough processors to attend the arriving application. When there are, RMS requests the processors launching *shrinkage actions* on the malleable applications. When processors are released, RMS can serve the arriving application with all required processors instead of enqueue it. This procedure is illustrated in Figure 2.2 (a). Notice that the new application must wait while processors are released, but its response time tends to be lower than when it must wait in the pending queue.

Completion of the applications: Processors are released when an application finishes. In general, if there are applications in the pending queue, RMS allocates these free processors to them, otherwise processors will remain idle.

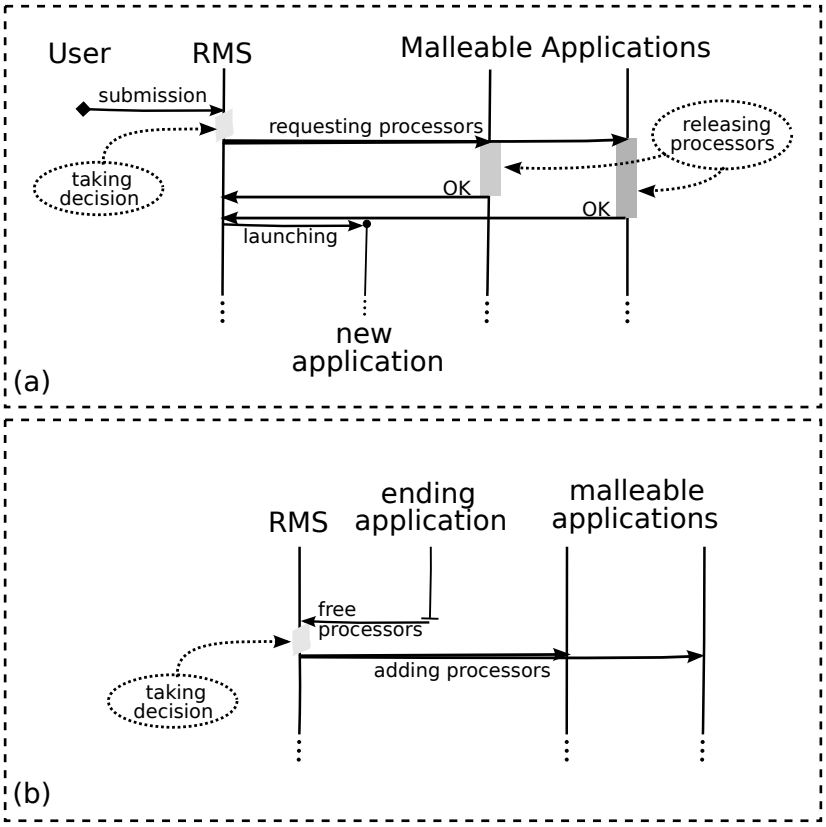


Figure 2.2: Use cases with volatile processors: (a) An user submit an application to RMS, which identifies that there are not enough processors to attend it and requests the missing processors from malleable applications; when the processors become free, the arriving application is served. (b) An application finalizes releasing some processors; RMS analyses the malleable application needs and decides to allocate the free processors to them.

When there are malleable applications, RMS may analyse their needs, verifying if they are able to receive more processors. In the affirmative case, the free processors are allocated to malleable applications launching *growth actions* on them. The interactions of this procedure are illustrated in Figure 2.2 (b). Increasing the number of processors of running malleable applications, it is expected improvements in their performance as well as in the utilization of the parallel environment resources (processors, bandwidth, and so on).

RMS that supports malleable applications can offer QoS (Quality of Service) advantages such as reductions in the response time, and improvements in resources utilization (UTRERA; CORBALÁN; LABARTA, 2004; HUNGERSHÖFER, 2004; BUISSON et al., 2007). However, to support it the RMS must have a scheduling policy able to deal with dynamicity. In other words, beyond its default behavior, the RMS must consider the needs of malleable applications (for example, the minimum and maximum number of processors) to takes its scheduling decisions. Furthermore, an acknowledge mechanism is desirable to confirm the successful execution of the malleable actions (growth or shrinkage), or even alert for some possible problem during the malleable actions.

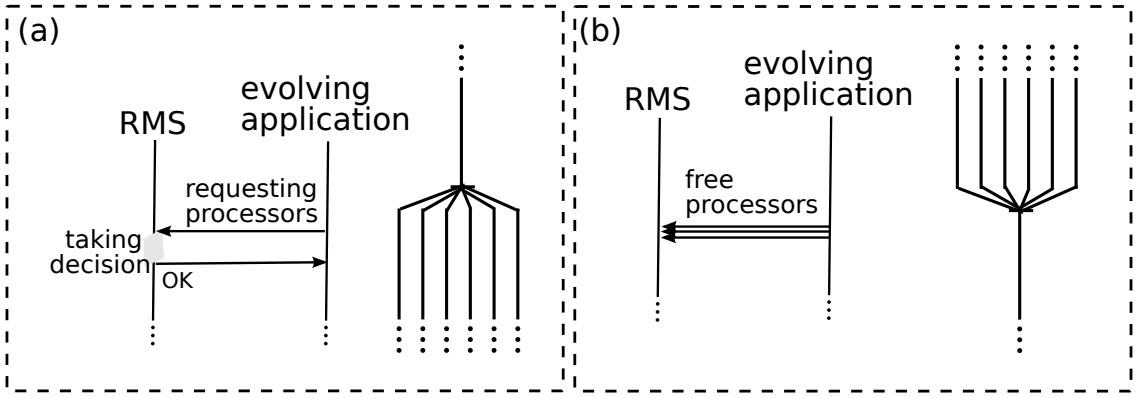


Figure 2.3: Use cases with unpredictable needs: (a) The evolving application increases workload and requires new processors to RMS, which takes the decision as the scheduling policy. In the right, we illustrate the increasing of the workload through dynamic generation of tasks. (b) The evolving application decreases workload and release some processors. In the right, the workload decreasing is illustrated through the tasks finalization.

Malleable applications must be able to identify events from RMS that launch malleable actions. After identifying these events, malleable applications must perform the coherent procedure, releasing or adding processors. Thus, the support of malleability requires an integration and cooperation between RMS and malleable applications (FEITELSON; RUDOLPH, 1996).

2.3.2 Use Cases with Unpredictable Needs

Nowadays, many problems require dynamic and irregular structures in their solution (AYGUADÉ et al., 2009), causing variations on the workload at runtime. Besides algorithmic features, dynamic variation in workload can be caused by the nature of the problems, and the unpredictable non-uniform distribution of the input data (GHAFOOR, 2007). Furthermore, the variations may be regular or not from one execution to another. For instance, for the same input, the N-body simulations solved with a parallel Fast Multipole algorithm will always have the same workload variations. On the other hand, weather prediction algorithms may require quick responses to sudden weather changes causing different workload variations from one execution to another. In both cases, unpredicted variations may change the application needs at runtime such as the number of processors, disk storage and network bandwidth. As introduced in Section 2.2, these applications are named as evolving and, we illustrate some use cases of them in the following.

We consider the typical scenario in which users submit applications to RMS, and it schedules them on the available processors using a scheduling policy. In addition, there is an evolving application running upon a set of processors. Thus, there are two situations in which unpredictable needs can require adaptive actions:

Increasing of the workload: When the workload of the applications increases, their demand for CPU, memory, disk space, bandwidth and so on, also increase. For instance, explicit task applications increase the workload when it unfolds the parallelism generating abstract tasks at runtime. There are two options to supply the extra demand of the evolving applications: either RMS

is able to offer at runtime more resources or the application schedules the further workload upon the existing set of resources, trying to reduce the impact of the irregularity. The first option involves hard issues such as to ensure that the environment will have enough resources to supply the application unpredictable needs. Figure 2.3 (a) shows the iterations between RMS and evolving applications as well as an illustration of increasing of the application workload. On the other hand, the second option tries to adapt the execution of the applications using their previously allocated resources. This may causes some loss of performance when compared with the first option, but it is independent of RMS issues and widely used in the current researches.

Reduction of the workload: When the workload decreases, some resources trend to have a low load or even become unused. Using explicit task applications as an example, the workload decreases when some abstract tasks finalize. If RMS provided extra resources to compute the further workload, when it ends, some processors become unused, and then they can be released (returned to RMS). Figure 2.3 (b) shows the iterations between RMS and evolving applications when the workload decreases as well as an illustration of the application workload decreasing. On the other hand, if the applications schedule the further workload among their previously allocated resources, when workload decreases, the distribution becomes unbalanced again. Thus, new workload redistribution is required to balance the load among the application resources.

As can be observed, the key issue to support evolving applications is to decide if RMS will be involved or not in the adaptive actions. An RMS scheduling policy that is able to deal with evolving applications is still hard due to their unpredictable nature and, as far as we know, was never provided. On the other hand, there are many studies aiming to provide load balancing on application side. Thus, we opt for this last approach.

2.4 Requirements to support Adaptability

In the previous section, we show use cases that require adaptive actions, *i.e.*, situations that requires adaptation: to execute with volatile processors; and to adapt the execution as unpredictable needs. During the explanations about the use cases, we already show some requirements to support them. In this section, the focus is to introduce further issues and requirements to support volatile processors - Section 2.4.1; and unpredictable needs - Section 2.4.2.

2.4.1 Requirements of Volatile Processors

Section 2.3.1 described the importance of RMS to execute malleable applications upon a set of volatile processors is clear. It decides when the adaptive actions must be launched, upon which applications, and involving how many processors. Furthermore, malleable applications have their own needs informed by the users at submission time. An example of needs is the range of processors, which determine that applications fail when there are fewer processors than the minimum, and otherwise, they can not improve performance using more processors than the maximum. Moreover, some problems have algorithmic restrictions requiring a specific number of processors on malleable actions, for example power of 2 processors.

Table 2.2: The main issues of environments with volatile processors and malleable applications.

Who requests adaptive actions?	The RMS.
When adaptive actions can be launched?	When new applications are submitted or running ones are finalized.
What guides the adaptive actions?	The RMS scheduling policy based on the availability of the processors, the malleable application needs, and the running and waiting queues.
What are the adaptive actions?	*Growth* - some workload is destined to new processors. *Shrinkage* - some processors are released without compromising the application results.
What are the main issues?	*Growth* - to select non-computed workload for new processors. *Shrinkage* - to ensure that the loss of processors will not affect the correctness of the results.

The design of an RMS scheduling policy able to deal with volatile resources is considered out of the scope of this thesis. Thus, we consider that there is an RMS able to manage the dynamic availability of the processors and deliver the decisions to malleable applications (more details about the management of volatile processors on RMS system will appear in Section 5.1). Thus, we can focus on the remaining issues such as establishing a communication channel between RMS and malleable applications, providing the identification of the dynamic events that launch the adaptive actions, designing the procedures that implement the malleable actions, and so on.

Malleable applications perform two adaptive actions: growth – to use more processors than currently exists, and shrinkage – to release processors. The growth includes a mechanism to delegate some workload to new processors. For instance, this mechanism may launch or migrate units of executions (processes or threads) to the new processors, according to the program design adopted. Thus, the challenge is to continue the application execution identifying non-computed work to be delegated at runtime. In consequence, this includes an additional care with correctness of application results.

On the other hand, the shrinkage procedure must ensure that the applications will continue running after losing some processors. As well as in growth, the implementation of this procedure depends on program design, in which the units of execution (UEs) running in releasing processors must be finalized or migrated. Moreover, the loss of processors should not compromise the application results. For instance, if an UE is finalized losing some already computed data, the application must be able to identify it and restart the computation in the future.

Table 2.2 summarize the most important issues about environments with volatile processors, and malleable applications.

Table 2.3: The main issues of supporting unpredictable needs of evolving applications.

Who requests adaptive actions?	The application.
When adaptive actions can be launched?	When the workload of the evolving applications increases or decreases.
What guides the adaptive actions?	The natural application irregularity.
What are the adaptive actions?	*Adapt to workload increasing* - to schedule the further workload among the units of execution. *Adapt to workload reduction* - to restore the load balancing among the units of execution.
What are the main issues?	Provide a scheduling strategy able to deal with unpredictable needs at runtime.

2.4.2 Requirements of Applications with Unpredictable Needs

Currently, explicit task applications (*i.e.*, those are developed following the explicit task parallelism programming paradigm), which are a kind of evolving application that execute without RMS interactions, have been widely studied. These applications are able to unfold the parallelism at runtime through the generation of the abstract tasks. Parallel APIs such as Cilk++ (LEISERSON, 2009), OpenMP (CHAPMAN; JOST; PAS, 2008) and TBB (REINDERS, 2007) (these APIs will be described in Section 3.2) use the explicit task parallelism to adapt the application execution to the required degree of parallelism on shared-memory environments (focused on multi-core architectures). In distributed-memory, KAAPI (GAUTIER; BESSERON; PIGEON, 2007) and AMPI (HUANG; LAWLOR; KALÉ, 2003) (will be also detailed in Section 3.2) can adapt their executions and scheduling to ensure performance taken into account the locality of the UEs on clusters of computers.

When users submit explicit task applications to RMS, the initial needs are described (number of processors, and so on). Although application can be launched based on these needs, they will become not ideal when the application workload varies. Thus, adaptive actions are required to balance the workload distribution and ensure efficiency. In this case, the adaptive actions involve some scheduling strategy to provide load balancing, and further issues related to the programming model. For instance, when tasks are generated at runtime (increasing of the workload), they must be assigned to units of execution (processors or threads). According to the program design, one unit of execution can compute only one task and finalize (probably tasks with a high granularity) or it receives a set of tasks maintaining a queue of them.

As the problems solved by evolving applications are naturally irregular, the workload tends to become unbalanced at runtime. Furthermore, when tasks are finished (decreasing of the workload) some processors keep without or with low load. Both situations require adaptive actions to rebalance the workload using an online scheduling strategy. We will present further issues of explicit task applications scheduling in Chapters 3 and 6.

Table 2.3 summarizes the main issues of these applications and their support on current parallel environments.

2.5 Program Structures and Adaptability

As introduced previously, adaptive algorithms are able to take adaptive decisions as resource availability or input data properties, both discovered at runtime (CUNG et al., 2006). Furthermore, a runtime adaptation requires some procedures that implement the coherent reactions when the set of resources is volatile or the input data causes variations the application workload. Moreover, these issues must be addressed by the program structures used by the adaptive applications. The question remaining is: *"What is the impact of an adaptive behavior according to the program structure used by the application?*

This section investigates answers to this question. It will introduce the program structures mostly used in parallel applications as well as their adoption by the related works involving adaptability. The program structures presented are: SPMD (Section 2.5.1), Master/Worker (Section 2.5.2), Loop Parallelism (Section 2.5.3), and Fork/Join (Section 2.5.4).

2.5.1 Single Program, Multiple Data

In Single Program, Multiple Data (SPMD) program structure, all units of execution compute the same program (Single Program) in parallel, but each UE has its own set of data (Multiple Data). The UEs can follow different paths in source code, which is determined by unique labels such as the process identifier (ID). This program structure is widely used to implement MPI applications, since it is flexible and covers the most important algorithm structures used in scientific computing (MATTSON; SANDERS; MASSINGILL, 2004). Figure 2.4 shows the basic elements of SPMD programs in which there are N UEs executing the same steps:

- As the parallel programming model, the UEs initialize;
- Each UE gets its unique process ID;
- Then, each UE computes the set of instructions on its own set of data, usually driven by the process ID;
- An UE finalizes when the set of data were computed.

The last two steps may also involve the distribution and recombination of the data according to environments features (shared or distributed memory) or programming paradigm (shared access or data replication).

The SPMD program structure appears in adaptive initiatives that use the MPI interface, because of its large use in scientific MPI applications. For example, the AMPI or Adaptive MPI (HUANG; LAWLOR; KALÉ, 2003) converts SPMD MPI programs in multi-partitioned ones, in which MPI processes are replaced by threads, encapsulated into Charm++ objects (KALÉ; KRISHNAN, 1993). AMPI programs are able to adapt themselves to imbalances in the workload of physical processors through the migration of Charm++ objects. Furthermore, they can adapt to volatile processors thanks to the Adaptive Job Scheduler (KALÉ; KUMAR; DESOUZA, 2000), which provides information about the processors availability to AMPI programs.

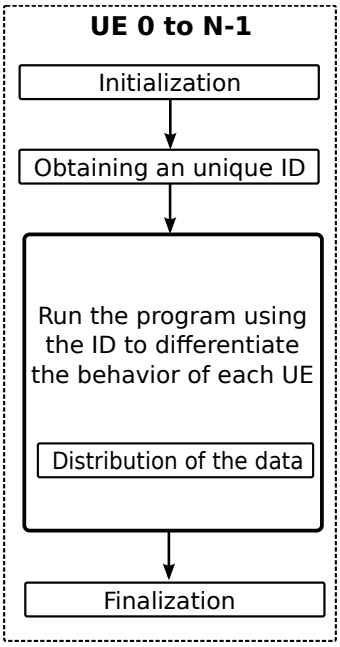


Figure 2.4: SPMD program structure: N UEs executing the same source code guided by their process ID.

Another initiative is Dynaco (BUISSON; ANDRÉ; PAZAT, 2007), which allows the execution of adaptive applications able to deal with volatile processors. In this initiative, malleability is a special case of adaptability. Dynaco is a generic framework that supports several different program structures. The AFPAC module (BUISSON; ANDRÉ; PAZAT, 2006) allows the execution of SPMD programs in Dynaco, and thus, allows that several MPI-based applications become malleable. The procedures implementing the adaptive actions are provided by the programmer to Dynaco, which is responsible to call these procedures properly.

Similarly, PCM (Process Checkpointing and Migration) (MAGHRAOUI et al., 2007, 2009) is a library that allows MPI applications to adapt themselves to dynamic changes in processors availability. Malleability is provided using process migration. Furthermore, the library controls the granularity of the processes, splitting or merging them and redistributing their input data aiming to ensure performance.

2.5.2 Master/Worker

In this program structure, there are two types of UEs: master and workers. The master starts the computation by setting up the problem and creating a bag-of-tasks. The workers take tasks from the bag, compute, and return the results when they are required. These steps are repeated as long as there are tasks in the bag. The master finalizes when it receives all worker results and the bag-of-tasks is empty. In classical implementations of Master/Worker programs, the master waits while the workers compute their tasks. Improved implementations keep some tasks to be computed by the master to avoid master idle standby. Furthermore, to reduce the communication costs, workers can take a cluster of tasks instead of take only one task at a time. In the same way, results may be returned in clusters instead of one result at a time.

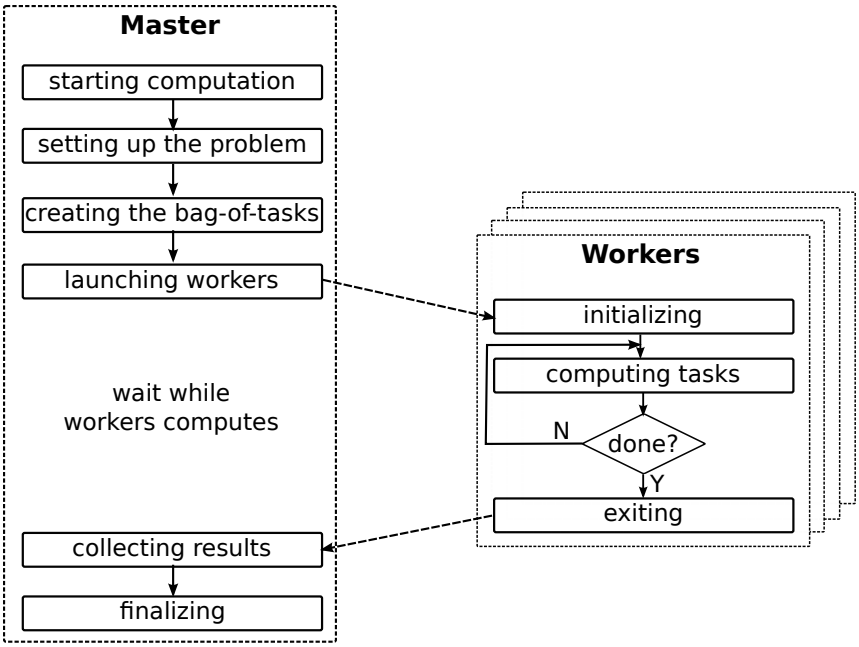


Figure 2.5: Master/Worker program structure: master starts the computation creating and managing tasks and workers; and workers take and compute tasks.

The Master/Worker program structure is usually adopted when there are no dependencies among tasks. Furthermore, this program structure offers an automatic load balancing among workers, since task assignment is made on demand. Although the master may be a bottleneck when the number of workers is large, in the average case this approach usually provides a satisfactory load balancing. Thus, there are many implementations of scientific problems following this problem structure. Figure 2.5 illustrates an implementation of a Master/Worker program:

- **Master:** starts the computation by setting up the problem, creating the bag-of-tasks, and launching the workers. It waits while workers compute and collects worker results. When the collection ends, the problem is considered solved and the master finalizes;
- **Workers:** Each worker initializes getting a cluster of tasks and while there are tasks, it computes them. When there is no more tasks, the worker sends results to master and ends its execution. Notice that workers only get task once, which is a decision that aims to reduce communication costs. However, this may be unefficient when the size of the cluster of the tasks is large.

Master/Worker program structure appears in adaptive initiatives because of its great popularity. For instance, PCM (MAGHRAOUI et al., 2009) also has results showing the adaptability of Master/Worker MPI programs allowing their execution upon a variable set of processors.

Another example is shown in LEOPOLD; SÜSS (2006); LEOPOLD; SÜSS; BREITBART (2006), which provides Master/Worker adaptive applications combining MPI-2 features and OpenMP threads. The target application is the WaterGAP (Water - Global Assessment and Prognosis), which investigates the current and future water availability worldwide. This application has irregular tasks of different sizes. It adapts itself according to the architecture setup (number of processors/-cores) combining process and thread creation, driven by the size of the tasks.

2.5.3 Loop Parallelism

The majority of scientific problems are overwhelmed with loop constructs. The loop parallelism program structure aims to transform a serial program composed by a set of compute-intensive loops into a parallel one, through the parallel execution of the loop iterations. This is an algorithmic strategy to provide a behavior similar to older vector supercomputing on modern parallel computers (MATTSON; SANDERS; MASSINGILL, 2004).

From a sequential loop-based program, loop parallelism program structure involves:

- **To find the most computationally intensive loops**, which dominate the program execution time, through an analysis of source code or performance of the serial program;
- **To eliminate dependencies among loop iterations** changing or adapting the source code, once loop iterations must be independent in loop parallelism;
- **To parallelize the loops** distributing the loop iterations among the UEs;
- **To optimize the scheduling of the loop iterations** aiming at load balancing among UEs.

Figure 2.6 shows an example of loop parallelism. There are two pseudo codes of a `for` loop, computing N independent iterations, being N predefined as 100. In the left, there is an iterative loop as well as an illustration of its execution. Notice that loop iterations are executed sequentially, one after another. In the right, the same `for` loop is shown with directive `# ParFor` that distributes the iterations among 10 UEs. The definition of the number of UEs involved is made direct in the runtime system. The 100 iterations are distributed among the UEs, where each performs 10 iterations. Notice that all iterations of the loop have the same workload.

To clarify the impact of loop parallelism in applications performance, suppose that to compute 1 iteration it is spent 1 second. Then, the iterative loop spends $100 \times 1 = 100$ seconds, whereas the parallel loop spends $10 \times 1 + parallel\_overhead = 10 + parallel\_overhead$ seconds. The *parallel_overhead* means the time spent to coordinate parallel tasks, such as task start-up and termination time, synchronizations, software overhead, etc. In an ideal system, for efficiency reasons, the *parallel_overhead* is less than the time spent in the parallel work. Then, we can estimate that $10 + parallel\_overhead > 10$ and $10 + parallel\_overhead < 10 + 10$. In other words, even in the worst case, the parallel loop reaches a better performance than an iterative execution.

An important issue of this program structure is that the loop parallelized must have enough parallelism (*i.e.* iterations with large computing time or a large number of fine-grain iterations) to compensate the overhead of parallelization. Thus, there are common practices such as: merge a sequence of loops with consistent loop limits into a more complex loop iteration; and to coalesce nested loops combining them into a single loop with a larger iteration counter. Notice that to allow the distribution of the iterations among the UEs, the loop parallelism program structure requires that the number of iterations are previously known. Furthermore, the parallel execution of iterations on a set of UEs requires some shared access of the input data. Thus, mostly implementations with this program structure are in shared-memory environments.

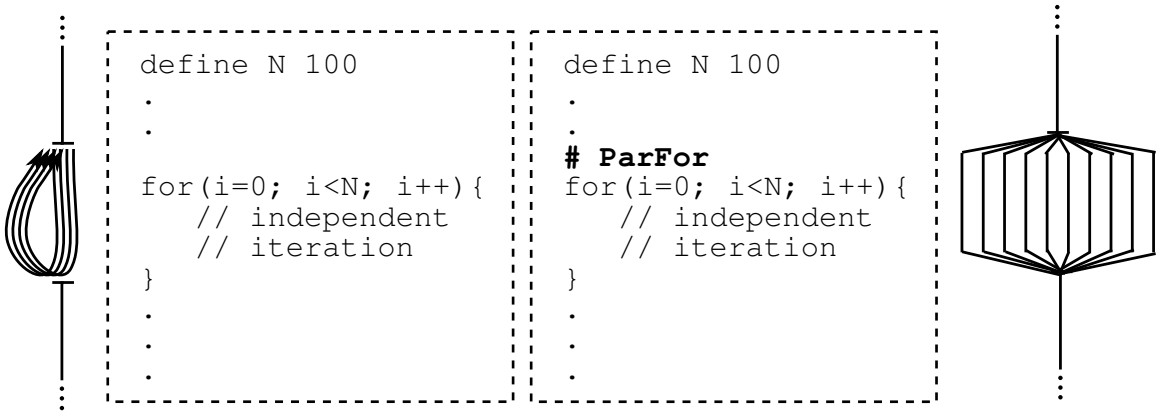


Figure 2.6: Iterative and parallel loop. In the left: an iterative loop with an illustration of its execution. In the right, the parallel loop (parallelism through `# ParFor`) using 10 UEs.

Loop parallelism became popular in the HPC community because of the large number of loop-based algorithms. OpenMP (CHAPMAN; JOST; PAS, 2008) was the first parallel programming API to support constructs to parallelize loops. It offers compiler directives to be included in the sequential source codes, signaling loops that must be parallelized by the OpenMP compiler. Thus, the assignment of iterations on UEs happens at compile time. Other examples of APIs that also provide loop parallelism are Cilk++ (LEISERSON, 2009), and TBB (Threading Building Blocks) (REINDERS, 2007) (more details about these APIs will appear in Section 3.2).

Although loop parallelism has a great acceptance on the HPC community, the natural restrictions of this program structure, blocks its use on adaptive applications. In fact, the units of work, or how many iterations each UE must perform, are statically defined at compile time. In addition, the total amount of loop iterations must be known at compile time and cannot be changed during the execution. These static features difficult the development of adaptive applications based on loop parallelism program structure.

2.5.4 Fork/Join

The Fork/Join program structure aims at problems that the algorithmic solution imposes a dynamic creation of tasks (to fork new tasks) and their termination (to join with the task that forked it), at runtime. Furthermore, Fork/Join programs preserve the relationship among tasks while they are managed. For instance, the recursive algorithms preserve the relationship between one level of recursion and another. This kind of algorithm can be implemented following the Fork/Join program structure: recursive calls fork new tasks, and when a task ends, it joins with its forker (MATTSON; SANDERS; MASSINGILL, 2004).

A popular use of Fork/Join is to implement the Divide and Conquer (D&C) algorithm structure. A problem is recursively split into subproblems until the computation become trivial (divide phase); the final solution is the merge of all subproblems solutions (conquer phase). Thus, the divide phase can be implemented by forking new tasks in each recursion level. In the conquer phase, a task can only join with its forker when its children have already joined it (hierarchical dependency).

```
 1. void DivideAndConquer ( type_problem problem ){
 2.   solution = divide( problem );
 3. }
 4.
 5. type_solution divide( type_problem problem )
 6. {
 7.   type_problem subProblems[2];
 8.   type_solution subSolution[2], solution;
 9.
10.   if ( problem is trivial )
11.   {
12.     solution = compute ( problem );
13.   }
14.   else
15.   {
16.     subProblem = split( problem );
17.     subSolution[0] = divide( subProblem[0] );
18.     subSolution[1] = divide( subProblem[1] );
19.     solution = conquer ( subSolution );
20.   }
21.   return solution;
22. }
23.
24. type_solution conquer ( type_solution subSolution ){
25.   return ( subSolution[0] + subSolution[1] );
26. }
```

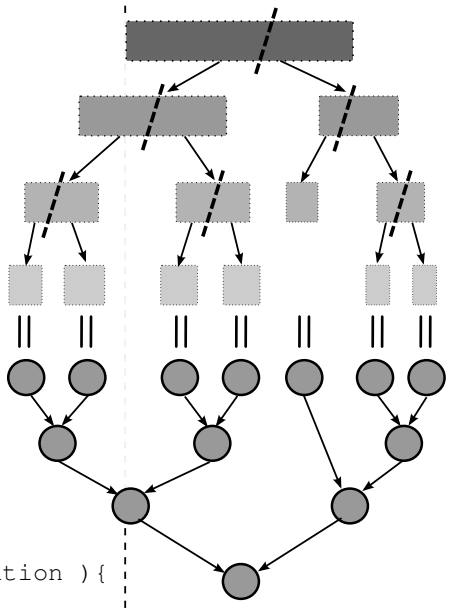


Figure 2.7: Pseudo code of a Divide and Conquer program (left side) and its parallel execution (right side).

Figure 2.7 shows a pseudo code of a D&C program and an illustration of its parallel execution. The `DivideAndConquer` function (line 1) starts the divide phase with the problem to be solved as parameter. The `divide` function (line 5) tests if the problem is already trivial: if yes, it is computed, otherwise the problem is split into two subproblems and, recursively, two `divide` calls are performed, one to each subproblem. The recursion only stops when the problem becomes trivial. When the subproblems solutions are available, they are merged through the `conquer` function (line 24). The illustration on the right shows the parallel execution of the pseudo code: on the top, the problem is split into subproblems until they become trivial representing the divide phase; then, the conquer phase will merge the subproblems solutions following their recursion level dependencies.

The Fork/Join program structure has a natural dynamic behavior since tasks are forked and joined at runtime. Furthermore, the generation of dynamic tasks is mostly, if not always, driven by the input data. For instance, in D&C applications, the problem is often represented by a set of input data. When this set becomes small enough to be computed efficiently, the divide phase stops. Thus, the Fork/Join program structure can be used to provide adaptability according to the properties of the input data.

There are many adaptive initiatives using the Fork/Join program structure. In shared-memory context, the Fork/Join is used in applications that follow the explicit task parallelism programming paradigm. For instance, Cilk (BLUMOFE et al., 1996; BENDER; RABIN, 2000) and its current release Cilk++ (LEISERSON, 2009), OpenMP 3.0 (AYGUADÉ et al., 2009), and TBB (REINDERS, 2007) (these initiatives will be detailed in Section 3.2). Leaving aside the technical differences, basically, all these approaches provide means for the programmers to define what

is a task and what are the dependencies among them. The runtime environment of these systems provide the dynamic generation of the tasks as well as their mapping and scheduling on UEs. In few words, the generation of the dynamic tasks at runtime represents the fork, while the satisfaction of the dependencies among tasks represents the join.

In the distributed-memory context, KAAPI (Kernel for Adaptative, Asynchronous Parallel and Interactive programming) (GAUTIER; BESSERON; PIGEON, 2007) has many similarities with Cilk and also uses the Fork/Join program structure. Furthermore, KAAPI considers locality issues on the scheduling, since it aims at distributed-memory systems such as clusters of computers and clusters of clusters.

2.6 Conclusion

The purpose of this chapter was to contextualize the adaptability needs in current parallel environments. First of all, we described the taxonomy that will be employed in this thesis. Then, we showed well-known classification of applications: rigid, moldable, malleable and evolving. Our thesis is focused on the last two classes which deal with volatile processors and irregular workload, respectively. We discussed about some use cases to show typical scenarios that require adaptability to allow efficiency. These use cases helped to compose a list of requirements to support volatile processors and irregular workload (Tables 2.2 and 2.3).

Furthermore, this chapter presented the mostly used program structures for parallel programming and their use in related works that provide adaptive applications. Table 2.4 resumes the four program structures described.

Table 2.4: Resume of the main features of the program structures and their use on related works on adaptability.

	Description	**Its use for adaptability**
SPMD	Copies of the same source code executing in parallel, computing their own data	In approaches based on MPI applications: AMPI, Dynaco, and PCM
Master/Worker	A master manages a bag-of-tasks and the workers compute tasks from this bag	PCM (MPI-based) and WatherGAP application (MPI-2 and OpenMP)
Loop Parallelism	To compute loop iterations in parallel	No related work on adaptability was found
Fork/Join	Algorithm requires to fork and join tasks at runtime	Based on explicit task parallelism: Cilk, TBB, OpenMP, and KAAPI

The key aspect of this chapter is that adaptive actions often require some support from the runtime environment. In this sense, the next chapter aims to detail this runtime support: *(i)* RMS and parallel applications interactions aiming to deal with volatile processors; *(ii)* Application Programming Interfaces (API) able to adapt the application execution at runtime; *(iii)* Scheduling issues required by adaptive applications aiming at load balancing.

3 RELATED WORKS: RUNTIME ISSUES OF ADAPTABILITY

Previously, we introduced the context of the current parallel environments and the scenarios that require adaptability. This chapter gathers initiatives that provide the execution of the adaptive parallel applications. The runtime environment issues that will be analysed are:

- The support of volatile processors by the RMS systems – Section 3.1;
- The APIs features that allow adapting the application execution to unpredictable needs – Section 3.2;
- The on-line scheduling required to deal with dynamicity at runtime – Section 3.3.

Thus, its goal is to provide a view of the related works aiming at issues involving the support of adaptability at the runtime environment level.

3.1 RMS Issues to support Volatile Processors

According to FEITELSON; RUDOLPH (1996), a good approach towards the support of the dynamic availability of processors requires a co-design of runtime system and programming environment. This statement refers the twofold issues of supporting volatile processors: on the one hand, the application must inform its basic needs and be able to adapt to variations in the processors availability (as well as to confirm the execution of the adaptive actions). On the other hand, RMS must be able to use the information about the basic needs of adaptive applications aiming at a good allocation of the processors. RMS can only demand changes in processors allocation when it can improve either the execution time of the running applications (add new processors) or the response time of a arriving applications (require some processors from running applications).

Thus, the support of volatile processors requires: *(i)* parallel programs able to deal with changes in the number of processors at runtime; *(ii)* interactions between the RMS and the programs aiming to exchange the required information; and *(iii)* specific scheduling policies to guide the RMS decisions about the allocations of the processors. In the following, there is an analysis of the treatment of these aspects in the related works.

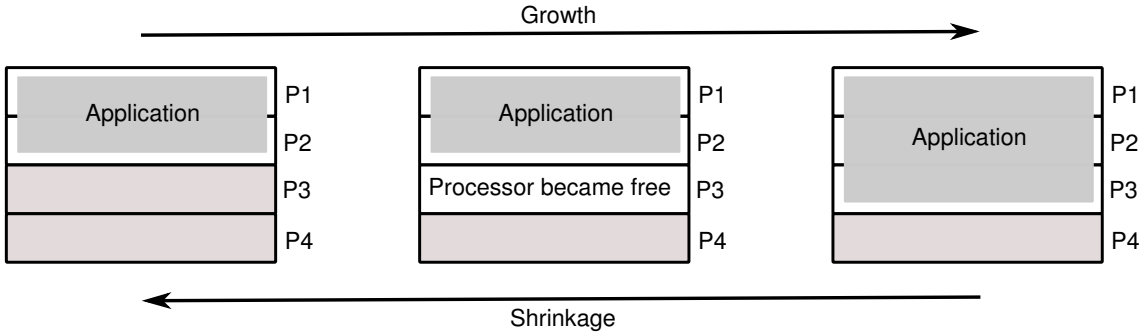


Figure 3.1: Growth and shrinkage actions of malleable applications: in growth (left to right) the application increases its number of used processors, while in shrinkage (right to left) it can release processors.

3.1.1 Implementation of the Adaptive Actions in Parallel Programs

In Section 2.3.1, were introduced the use cases involving volatile processors and the actions required to allow an on-the-fly adaptation to them. Applications able to deal with volatile processors are named as malleable, and basically, they have two special procedures:

- Growth action to allow the application to use new processors;
- Shrinkage action to allow the application to release some processors.

Figure 3.1 illustrates these adaptive actions. In the following, we will analyse how to they are implemented by the related works.

3.1.1.1 ***PCM (Process Checkpointing and Migration)***

PCM (MAGHRAOUI et al., 2007; DESELL; MAGHRAOUI; VARELA, 2007; MAGHRAOUI et al., 2009) includes an extension to provide malleability as a user-level library for iterative MPI applications. PCM applications can reconfigure dynamically to changes in the availability of the processors through: *(i)* split and merge operations - to change the number of running processes; and *(ii)* processes migration - to change the locality of the processes. Split and merge allow any number of processes M to be split or merged into any number of processes N. The PCM library offers four primitives to implement these operations: two for operations 1 to N (`PCM_Split` and `PCM_Merge`), and two for M to N (`PCM_Split_Collective` and `PCM_Merge_Collective`).

Basically, a set of PCM high-level primitives provide low-level issues for programmers. For example, they implement changes in the rank and number of the MPI processes, as well as offer a malleable communicator called `PCM_COMM_WORLD` that replaces the standard global communicator `MPI_COMM_WORLD` (more detail of this standard communicator will appear in Section 4.1.1) As these operations change the communication topology, they also requires data redistribution, which is provided by the library. On the other hand, PCM requires that the programmers include the PCM primitives into the MPI source code and specifies the data structures involved in adaptive actions.

While split and merge operations change the communication topology and data distribution, the processes migration changes the mapping of the processes into physical resources (MAGHRAOUI et al., 2009). The migration adjusts the process-level granularity providing more scalable and flexible reconfigurations. PCM process migration is implemented using `MPI_Comm_spawn` to spawn new processes in the target processors. A new process receives a checkpoint of the local data of the migrating process (the spawner), and continues the computation of this local data. Since the migration changes the location of the processes, the entire application must known that changes are being performed. Thus, the PCM process migration is a collective operation that blocks all processes of the application, until the end of the transfer operation.

3.1.1.2 ***Adaptive MPI (AMPI)***

AMPI (HUANG; LAWLOR; KALÉ, 2003) also takes advantage of migration to adapt applications to changes in processors availability. In this case, Charm++ objects (which encapsulate threads that replaces the MPI processes) are transfered in the adaptive actions. Since parallel applications were implemented according to the AMPI specification, the adaptive actions are automatic and transparent. In other words, once Charm++ objects can be instantiated, the AMPI environment provides the migration of them, and thus, can adapt the application to changes in the processors availability. Thus, when new processors become available some objects are migrated to them. Otherwise, objects are migrated from processors that are being released. This behavior avoids failures while the application performs with volatile processors.

3.1.1.3 ***Dynaco***

Dynaco (BUISSON; ANDRÉ; PAZAT, 2005; BUISSON et al., 2007) provides adaptive applications through a special module AFPAC which allows malleability on MPI-Based ones. Dynaco applications perform event-based adaptations according to the dynamicity of the processors. When an application observes that new processors are available, it may increase the parallel degree by spawning new processes (using MPI-2 dynamic process creation: `MPI_Comm_spawn`). Otherwise, when processors are requested, the processes that run on them are terminated (BUISSON; ANDRÉ; PAZAT, 2005). To provide adaptability, Dynaco have four components:

- *Observe:* to monitor the environment and launch adaptive actions when some relevant change happens;
- *Decide:* to find the best strategy that an application should use to adapt itself;
- *Plan:* to determine the set of actions that implement the strategy chosen in the decide component;
- *Execute:* to schedule the actions listed in the plan.

The components described above compose a generic mechanism to provide adaptability, but it does not consider application-specific issues. For instance, which adaptive strategies are the most appropriated to a given class of problems; what are the required actions to implement these adaptive strategies; what is the impact of these actions on the application execution; etc. These issues cannot be solved without involve the programmers, which must provide to Dynaco the decision pro-

cedure, the description of planning problems, and the implementation of adaptation actions (BUISSON et al., 2007). Furthermore, if the adaptations occur while data are being modified, it may cause inconsistencies on the application results. Thus, Dynaco offers special states called *points* or *annotations*, which identifies in the application source code, when adaptive actions can be performed without compromising the results. The insertion of points in the source code is another responsibility of the programmers that use Dynaco.

3.1.1.4 ***Strength and Weakness***

Table 3.2 presents the strength and weakness of initiatives previously discussed.

Table 3.1: Strength and weakness of initiatives to support adaptive actions in the programming context.

Initiatives	Strength	Weakness
PCM	It allows to control the process granularity, changing the number of running processes and their locality (through migration)	Programmers must known deeply PCM primitives to be able to insert them in their source codes
AMPI	Charm++ objects are automatic and transparently migrated to implement adaptive actions	Application development must follows AMPI specification
Dynaco	It is a robust framework able to provide adaptability taking advantage of the MPI-2 features	Programmers must provide the decisions, planning and to implement the adaptive actions

3.1.2 Communication between RMS and Adaptive Applications

Adaptive actions able to deal with volatile processors require updated information about the availability of the processors. Malleable applications expect to receive this information from a component of the runtime system. The Resource Management Systems (RMS) usually has means to offer it, and many initiatives stablish a communication channel between the RMS and the malleable applications, as shown in Figure 3.2. In the following, we describe the exchanging mechanism in the related works.

3.1.2.1 ***PCM***

PCM receives information about the availability of the processes from a middleware – the IOS (Internet Operating System) (MAGHRAOUI; SZYMANSKI; VARELA, 2006). It is composed by a peer-to-peer network of agents performing as a specific RMS. IOS has three modules:

- Profiling to offer resource-level and program-level profiling;
- Decision to take reconfiguration decisions;
- Protocol to start Work Stealing requests.

The information collected in the program-level profiling is: processing, communication, data accesses and memory usage, whereas in resource-level profiling are: CPU processing power, memory, disk storage, and network bandwidth and latencies.

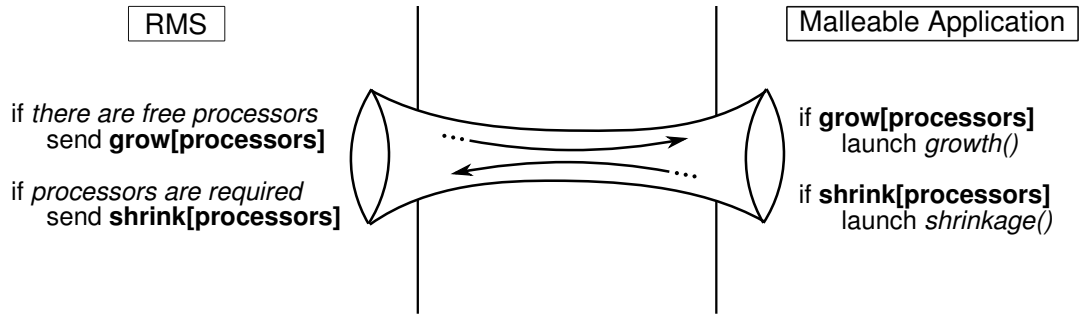


Figure 3.2: Communication channel between RMS and malleable application: RMS decides based on processors availability informing its decision to the application.

These information guide decisions about reconfiguration, which will be described in Section 3.1.3. The IOS agents perform Work Stealing requests to start reconfigurations every time they receive a notification that there are new available processors or when the existing ones become idle.

Through PCM, MPI programs and IOS interact using two interfaces: one to allow the IOS to profile the MPI programs, and another to transmit the reconfiguration decisions from IOS to MPI programs. The first interface is the IOS profiling API, which defines a set of functions to collect the current state of programs communications, and manage the profiling. The second interface uses a set of global flags in PCMD (PCM Daemons). They are daemons running on each processor to handle the checkpoint services and forward the reconfiguration requests. A master process (chosen among the MPI processes) probes the PCMD and broadcasts the reconfiguration notifications to the remaining processes. The status (`PCM_Status`) that can be reached are: *(i)* `PCM_MIGRATE` when processes need to be migrated; *(ii)* `PCM_RECONFIGURE` to notify that reconfigurations are being done; *(iii)* `PCM_SPLIT` when split operations are being done; *(iv)* `PCM_MERGE` when merge operations are being done.

3.1.2.2 ***AMPI***

AMPI applications receive information about the availability of the processors through the Adaptive Job Scheduler - AJS (KALÉ; KUMAR; DESOUZA, 2000). It allocates the AMPI applications, manages the use of the processors, ensures the migration of Charm++ objects, and keeps the system Quality of Service (QoS).

Every time that the AJS communicates with running AMPI applications, it happens via CCS (Converse Client Server) protocol. Communications are restricted to a bit vector transmission, which describes the command required. This bit vector is an easy manner to describe changes in the set of processors available to an AMPI application: the positions set by 1 are processors currently available. Thus, the AJS decides how many processes can be used by an application, sets up the bit vector and sends it to the application. Based on this vector, the application reconfigures itself migrating Charm++ objects when necessary.

3.1.2.3 ***Dynaco***

Dynaco works together with the Koala multi-cluster resource manager (BUISSON; ANDRÉ; PAZAT, 2007), which has specific mechanisms for data and processor co-allocation, resource monitoring and fault tolerance. Interactions between Koala and Dynaco happen through the Malleable Runner (MRunner). This runner includes a complete instance of Dynaco, which allows the handling of adaptive actions over the application. The MRunner receives notifications from Koala indicating changes in the availability of the processors. These notifications are messages with grow and shrink contents, which are propagated and translated into the appropriate adaptive actions. When the adaptation ends, the MRunner sends back acknowledgments to Koala confirming the adaptation to the new configuration of the processors.

3.1.2.4 ***Strength and Weakness***

Table 3.2 exposes the strength and weakness of communication mechanisms implemented by the initiatives previously discussed.

Table 3.2: Strength and weakness of communication mechanisms of the related works.

Initiatives	Strength	Weakness
PCM	The application probes and receives reconfiguration notifications through global flags	Centralized solution (master processes) which tends to lose efficiency in large scale environments
AMPI	Runtime transmit the availability of the processors through a bit vector (simple and easy)	Few changes in bit vector requires its retransmission and reconfigurations
Dynaco	Based on two phase commit protocol: notifications are sent by Koala and the adaptations must be confirmed	This protocol can block waiting for agreements

3.1.3 Scheduling Policies to deal with Volatile Processors

RMS decisions are the key issue to support malleability. These decisions must take into account processors availability and the applications requirements. This section exposes the policies found in related works to manage volatile processors.

3.1.3.1 ***PCM***

On the PCM approach, the IOS follows three policies: Transfer, Split and Merge. The first policy purpose is to determine when to transfer load from one processor to another and how much load must be transfered. The load of a processor is proportional to the number of running processes and the amount of CPU-processing power used. Based on the IOS peer-to-peer network, the Transfer policy tries to adjust the load between two peers comparing load measures and calculating the number of processes that must be transfered. Furthermore, IOS uses a heuristic to pick the best candidates to migrate from one processor to another.

The Transfer policy can only be efficient when there are enough processes to be migrated, adjusting the load balancing. Otherwise, the running processes can be split to allow the adjustment of the load. The Split policy guides this operation taking into account the percentage of the used processing power and the current processing power of both peers involved. The most loaded processes are split into fine grained ones so they can be migrated following the rules of Transfer policy.

On the other hand, when a node has a large number of running processes causing a large rate of operating system context switching, PCM can decide to merge some processes. The merge is a local operation guided by the Merge policy. This policy uses information about the rates of context-switching, a history of CPU utilization measures, and a measure of the stability of the surrounding environment (to verify if the neighborhood is stable or reconfiguring often). Thus, PCM decides to merge processes when it can improve the performance of the processor, the surrounding environment is stable and the context-switching rate is higher than a given threshold.

3.1.3.2 ***AMPI***

In AMPI, the adaptive decisions are taken when new applications are submitted. The Adaptive Job Scheduler policy considers the priority of the AMPI applications and the range (minimum and maximum number) of processors required by them. Thus, the policy first tries to allocate the maximum number of processors required by an AMPI application, even if AJS needs to launch shrinkage actions on lower priority applications to achieve enough processors. If the maximum number of processors cannot be achieved, AJS tries again aiming at the minimum number of processors. When these two attempts fail, the arriving AMPI application is insert in a queue of pending applications. Furthermore, the applications in the pending queue have their priority increased aiming to avoid starvation.

3.1.3.3 ***Dynaco***

In Dynaco, a malleable application starts to execute upon a set of processors allocated by Koala, which represents at least the minimum number of required processors. During the allocation, Koala follows one of its scheduling policies, such as the Worst-Fit, Close-to-Files, Cluster Minimization, and so on (BUISSON et al., 2007). These policies are common used to allocate processors by RMS systems.

Koala will decide about adaptive actions either when processors become available or when a job can only start if the malleable applications release some processors. It has two decision policies to guide adaptive actions: Favour Previously Started Malleable Applications (FPSMA) or Equi-Grow & Shrink (EGS). The FPSMA starts to grow from the earliest malleable application and shrink from the latest one. Thus the previously started application is favored by receiving new processors or not releasing them. The EGS policy aims to balance the distribution of the processes among malleable applications. Thus, the allocations or releases occur equally on all running malleable applications.

3.1.4 Summary of Adaptive Actions Implementations

Table 3.3 summarizes the issues on supporting volatile processors: the implementation of adaptive actions; the interactions between RMS and malleable applications; and the specific scheduling policies.

Table 3.3: Issues on support volatile processors as the related works.

	Adaptive Actions	**Interactions**	**Scheduling Policies**
PCM	MPI processes can be split, merged or migrated	IOS profiles the applications, which probe PCMDs to know IOS decision	Transfer, Split and Merge policies
AMPI	Based on Charm++ objects migration	AJS sends (via CCS) a bit vector describing reconfigurations to applications	Adaptive actions are launched when new applications arrive
Dynaco	MPI processes can be spawned or terminated	Koala monitors the processors usage and sends its adaptive decisions to MRunner	Favour Previously Started Malleable Applications and Equi-Grow & Shrink

3.2 APIs to deal with Unpredictable Needs of the Applications

While Section 3.1 focused on issues of the related works to support volatile processors, this section aims to analyse the initiatives able to deal with unpredictable needs of the applications. As exposed in Chapter 2, applications with unpredictable needs can be classified as evolving. This thesis will be focused on a particular subclass of them, the explicit task parallel ones or those who follow the explicit task parallelism paradigm.

The inherent irregularity of these applications requires some support in the programming interface to allow coherent reactions to adapt to unpredictable needs. Moreover, according to MATTSON; SANDERS; MASSINGILL (2004), the development of explicit task parallel programs involves three main aspects:

- The definition of abstract tasks;
- Means to solve dependencies among tasks;
- The on-line scheduling of the abtract tasks.

These aspects are analysed in some APIs, which are divided on distributed and shared-memory contexts.

3.2.1 APIs for Distributed-Memory Environments

This section exposes two approaches: AMPI (Adaptive MPI) that was already described in the previous section, and KAAPI (Kernel for Adaptative, Asynchronous Parallel and Interactive programming) (GAUTIER; BESSERON; PIGEON, 2007). Although these APIs have not been named for evolving applications (neither for explicit task parallelism) their adaptive nature fits the main requirements of this kind of application. Thus, we consider relevant to introduce their features in the context of this document.

The approaches will be described according to the definition of tasks, the dependencies among tasks, and the on-line scheduling decisions.

3.2.1.1 ***Definition of tasks***

AMPI programs are MPI ones converted through the Charm++ compiler. This translation means to convert MPI processes into user-level threads that are encapsulated on Charm++ objects, together with the input data. The computation of a program is divided into a large number of virtual processors (or Charm++ objects) that are assigned and scheduled to physical processors. The programmer still follows the MPI development design, but without being restricted by a number of physical processors. Thus, the partitioning of the workload can be more flexible aiming to best fit the nature of the parallel problem (HUANG et al., 2006). Furthermore, AMPI ensures performance assigning multiple fine-grained Charm++ objects to the physical processors, and thus, the AMPI Load Balancer (LB) can provide a fine calibration of the processor utilization, transferring some objects when necessary. Notice that AMPI already includes the guiding principle of the explicit task parallelism paradigm: extract all potential parallelism of the algorithmic structures defining abstract tasks (units of work), and leave the scheduling of the tasks on account of the runtime system.

In **KAAPI**, tasks are functions calls, which return no value except through the access of a global address space called global memory (GAUTIER; BESSERON; PIGEON, 2007). To allow the creation of tasks at runtime, KAAPI has in its high-level programming interface a special keyword `fork`, which is included in the application source code before a function call. To Illustrate the use of this keyword, consider a recursive implementation of the Fibonacci computation, in which the n^{th} Fibonacci number is the sum of the previous two ($n-1$ and $n-2$). Using KAAPI, the source code line "`fork fibonacci(n-1, res1);`" will create a new task to compute Fibonacci for `n-1` and the result will be stored in `res1`.

3.2.1.2 ***Dependencies among tasks***

In **AMPI**, dependencies among tasks can be understood as information that must be exchanged among Charm++ objects. Thus, all communications of AMPI applications are powered by Charm++ optimizations. For instance, using asynchronous communications to overlap with computations the time that a CPU waits for the end of data transferences; to cluster small messages; send several large messages in sequence; and special optimizations for collective communications defined at application starting time. Furthermore, as AMPI allows the migration of Charm++ objects, the library also provides messages forward to ensure that they will be delivery even after migration.

KAAPI solves dependencies through shared access of objects in the KAAPI global address space or global memory. To declare objects on the global memory, KAAPI includes another keyword, `shared`. Using the Fibonacci example again, "`shared int res1;`" declares the variable `res1` in the global memory. This variable will store the result of the computation: sum the result of Fibonacci to $n-1$ and $n-2$. Furthermore, the function signature should specify the access mode of the shared objects – read, write or access. For instance, the signature of the Fibonacci function would be "`void fibonacci(int n, shared_w int res)`", in which the `res` has a write access mode.

3.2.1.3 ***On-line Scheduling Decisions***

AMPI scheduling decisions are based on physical processors utilization. Its Load Balancer (LB) component collects information about the utilization of processors and object-communication pattern in background. When a balancing of the load is required, LB redistributes the workload based on collected information. In redistribution, objects from overloaded processors are migrated to underloaded ones (HUANG et al., 2006). Notice that the fine-grain nature of the AMPI applications allows achieving an efficient load distribution.

KAAPI program are composed by a dynamic set of processes. Each process is composed by several threads which execute KAAPI tasks. A runtime scheduler associates threads to a fixed number of virtual processors (called kernel threads), which are scheduled on physical processors. The load balancing of the KAAPI applications is based on the Work Stealing strategy: idle threads steal tasks from busy ones (more details in Section 3.3). There are two improvements in the strategy: KAAPI implementation of the Work Stealing favors local stealing than remote ones; and the stealing requests also are launched when a thread is blocked and there are no others ready to compute (improvement in the thread scheduler).

3.2.2 APIs for Shared-Memory Environments

This section aims to introduce the shared-memory APIs able deal with irregular problems. Three APIs will be described: Cilk (BLUMOFE et al., 1996; BENDER; RABIN, 2000), OpenMP 3.0 (AYGUADÉ et al., 2009; CHAPMAN; JOST; PAS, 2008), and TBB (Threading Building Blocks) (REINDERS, 2007). They also will be described according to the definition of tasks, the dependencies among tasks, and the on-line scheduling decisions.

3.2.2.1 ***Definition of tasks***

Cilk is a C-based multi-threaded runtime system design at MIT (Massachusetts Institute of Technology). It guides many other researches thanks to its theoretical and practical results proving its efficiency on shared-memory environments. Recently, Cilk has a C++ version, Cilk++ (LEISERSON, 2009), that provides some abstractions to programming using threads aiming at multi-core architectures (LEISERSON; MIRMAN, 2008). Basically, a Cilk program is a collection of procedures that are broken into a sequence of threads (BLUMOFE et al., 1996). Cilk has two keywords to allow the definition of the tasks: `cilk` to declare functions that can be performed by threads; `spawn` to call functions declared with `cilk` aiming to create a new thread. The Cilk's programmers define the abstract tasks using the `cilk`

keyword and they are dynamically created when the functions preceded by `spawn` are executed. Thus, the Cilk tasks can be seen as the workload computed by each thread, which is defined in program-level as functions.

OpenMP is a standard API for shared-memory platforms (CHAPMAN; JOST; PAS, 2008). It is composed by a set of compiler directives, library routines, and environment variables that influence the application behavior at runtime. The latest version of OpenMP, version 3.0, includes a tasking model which aims to simplify the development of parallel programs. Thus, the units of work are specified as *explicit tasks* within structured blocks, termed as task regions (AYGUADÉ et al., 2009). The programmer identifies the potential parallelism of the algorithms inserting the `task` directive in the source code, which defines the task regions. Furthermore, OpenMP programs include the `parallel` construct, which defines the parallel region of the source code. Thus, task regions are inside of parallel ones.

Intel© *Threading Building Blocks* **TBB** (REINDERS, 2007) is a C++ runtime library for multi-core architectures. TBB programs are developed in terms of tasks and the runtime library has full responsibility of tasks scheduling. It defines tasks as abstract units of work on user-defined classes, which derive from the abstract class `tbb::task`. In general, the concept behind the TBB programs is similar to the previous approaches: instead of develop parallel programs aiming to distribute the workload among a set of threads, the programmer identifies the potential parallelism of the algorithm and leaves the TBB runtime maps tasks to threads as well as provides load balancing.

3.2.2.2 ***Dependencies among tasks***

Cilk represents the computation in DAGs (Directed Acyclic Graph), in which the vertices are the threads and the edges are the relationship among them: a parent thread creates a child; a thread creates a successor (thread to wait for children results); and the child returns results to its parent. During the execution of a Cilk program, its DAG is dynamically unfolded. Cilk programs are fully strict (or well-structured): communications only occur between parent and children threads. Thus, the dependencies among tasks on Cilk are restricted to parent/children information exchanges. Cilk offers the `sync` keyword to force synchronization. Thus, through `sync`, the programmer identifies points from which the execution can only continue when the dependencies have been solved. In other works, the points in which the parent must wait for children to be able to compute the next instructions.

OpenMP deals with dependencies among tasks through the `taskwait` construct. It suspends the current task execution until all its children have completed. Furthermore, tasks automatically synchronize always that a parallel block ends. Thus, OpenMP has explicit barriers - set by the programmer through the `taskwait` construct, and implicit ones - default synchronization in the ending of parallel blocks.

In **TBB**, dependencies can be expressed using one of the two styles: *blocking* or *continuation passing*. In *blocking* style, the parent task must set the number of children to be waited with `tbb::task::set_ref_count` method. Thus, the parent blocks waiting for the execution of these children. The *continuation passing* style is similar to the Cilk *successor* concept in which an additional task is forked to receive children returns while the parent continues its execution.

3.2.2.3 ***On-line Scheduling Decisions***

Cilk provides an efficient scheduling of the workload among threads thanks to Work Stealing (BLUMOFE; LEISERSON, 1999). Each processor has a deque (double-ended queue) of ready tasks and when one remains without work it chooses randomly another one to steal some tasks. Further information about Work Stealing will be described in Section 3.3, however the key issue is that the Cilk scheduling is driven by consumption of the tasks on demand.

OpenMP tasks are scheduled into a team of threads, which are started by the `parallel` directive. By default, the execution of a task is tied with the same thread, but without implying in a continuous execution. A thread can suspend a task execution and resume it later according to scheduling requirements, when a scheduling point is reached. In this case, scheduling points are the `task`, `taskwait`, explicit or implicit barrier construct, and upon completion of the task. Scheduling decisions on OpenMP are based on several simple strategies that achieve satisfactory performance. However, the tasking model is new on OpenMP and it is still under development to include more complex strategies and some way to allow the programmer to handle the scheduling.

TBB also has a task scheduler based on Work Stealing (BLUMOFE; LEISERSON, 1999). It maps tasks to native threads for the most efficient usage of the underlying hardware, aiming to minimize memory demands and cross-thread communication. Hence, the runtime takes responsibility of scheduling for locality and load balancing. Furthermore, usually TBB tasks include few instructions; since the goal is to define tasks with the smallest grain possible and leaves the runtime scheduler provide the best fit of them among the current threads.

3.2.3 Summary of the APIs for Irregular Problems

Table 3.4 summarizes the main aspects of the APIs able to deal with unpredictable changes in application workload: how they define tasks; how they define and solve dependencies among tasks; and what are the mechanisms used to take scheduling decisions.

3.3 Scheduling of Adaptive Applications

Scheduling is a key issue to provide efficient adaptive applications since it guides the adaptive actions and allows balancing the workload distribution among parallel applications UEs. This section aims to show the main aspects of the scheduling of the adaptive applications found in the related works. Thus, Section 3.3.1 regards the efforts to schedule or map adaptive jobs according to their initial requirements. Afterward, Section 3.3.2 describes the scheduling strategy mostly used to provide load balancing to adaptive applications - the Work Stealing.

Table 3.4: APIs able to deal with irregular workloads in the related works.

	Definition of tasks	**Dependencies**	**Scheduling**
Distributed-Memory			
AMPI	Fine-grained tasks defined as MPI design	Optimized communications among Charm++ objects	Based on workload measures
KAAPI	Function calls preceded by `fork`	Accessing objects in a shared memory region	Work Stealing
Shared-Memory			
Cilk	Function calls preceded by `spawn` creating new threads	Parent/children relationship (`sync`)	Work Stealing
OpenMP	Units of work defined as structured blocks (`task`)	Explicit barriers (`taskwait`) and implicit ones (end of structured blocks)	Several simple strategies (still under development)
TBB	User-defined classes deriving from `tbb::task` class	Blocking style: parent blocks waiting for children; and continuation passing: forks a further task to wait	Work Stealing

3.3.1 Starting Time Scheduling

Scheduling is a challenge that motivates many researches in different fields. In parallel programming, the inherent complexity of the applications and the constant evolution of the parallel architectures require efficient scheduling solutions to ensure performance. Adaptive applications have extra scheduling complexities: the decisions must supply the basic requirements of the applications as well as provide the best utilization of the overall parallel architecture.

In this sense, many theoretical studies have been focused on scheduling strategies for adaptability (LEPÈRE; TRYSTRAM; WOEGINGER, 2002; DUTOT; MOUNIÉ; TRYSTRAM, 2004; DUTOT et al., 2005; JANSEN; ZHANG, 2005). The common goal of these studies is to minimize: the makespan (*i.e.*, the time spent in the scheduling); the average completion time (*i.e.*, assigning weights to applications and executing first the smallest); the response time; and the waiting time. All these cited studies provide efficient theoretical results using approximation algorithms. These algorithms divide the problem of scheduling malleable applications into two phases: allotment and makespan problems. In the first, the algorithm allocates as many processors as allowing an efficient execution of the malleable applications. In the second phase, scheduling algorithms for non-malleable applications are adopted, aiming to provide an efficient distribution of the workload. Although the name *malleable* was used in these studies, according to Feitelson and Rudolph classification these applications are *moldable* (see Section 2.2), because the number of processors is determined at starting time and does not change during the application execution.

Section 3.1.3 discussed the RMS scheduling policies able to deal with volatile processors and the focus was the on-the-fly decisions required to provide adaptability.

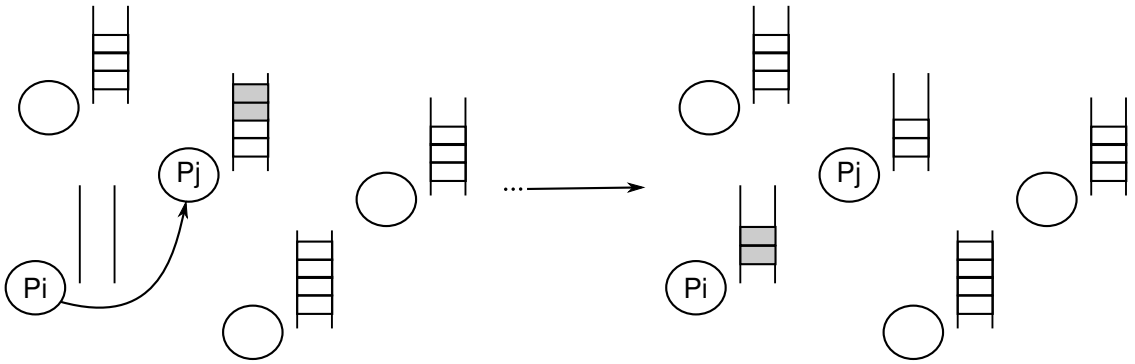


Figure 3.3: Work Stealing Strategy: circles represent the processors in which each one has its own queue of tasks (deque). The `Pi` processor becomes without tasks in its deque, then it is a thief. `Pj` is randomly chosen as a victim and `Pi` steals tasks from the top of `Pj`'s deque.

However, it may be observed that adaptability is used aiming at different goals. For instance, Adaptive MPI (AMPI) takes decisions based on requirements of the arriving applications, launching adaptive actions aiming to reduce their overall response time. Another use of the adaptability is to maximize the processors utilization. In this case, adaptive applications run in low priority together with rigid or moldable ones. Thus, they compute upon the free processors of a parallel architecture, *i.e.* those who are not used by other applications, improving the overall utilization of the resources (HUNGERSHÖFER, 2004; BUISSON et al., 2007; CERA et al., 2010).

3.3.2 Scheduling at Runtime

Along the previous sections, we cited Work Stealing as a large used strategy to provide efficient load balancing. In this section we will introduce some details about the strategy and its different implementations.

BLUMOFE; LEISERSON (1998) devised the Work Stealing to schedule Cilk tasks: each processor has a deque of ready tasks (named closure) and it inserts and consumes tasks from the bottom of the deque. When a deque is empty, it (the thief) chooses randomly another processor (the victim) to steal tasks from the top of victim's deque. Figure 3.3 illustrate this strategy. This algorithm is proved as efficient in homogeneous environments for fully strict programs in terms of time, space and communication (BLUMOFE et al., 1996). The proof relies on the random choice of the victim and on the homogeneous distribution of the stealing requests. Moreover, as Cilk has design for shared-memory, the cost of the thefts is independent of processors physical location.

Another important feature of Cilk's Work Stealing is the consumption of the tasks from the ready deque. Processors insert and consume tasks from the bottom and steal from the top. The reason to steal from the top is to allow the stealing of large amounts of work, since tasks in the top tend to execute longer than those in the bottom (remembering that Cilk tasks have a recursive nature). Thus, the unbalancing tends to be solved with few steals and the cost of the scheduling causes a low impact on the program performance. Moreover, tasks in the top of the deque are also in the top of the DAG, and the computation of the thief will make progress on the program critical path (BLUMOFE et al., 1996).

BENDER; RABIN (2000) proposed an enhanced scheduling for processors of different speeds. Basically, during a random steal, if the deque of the victim is empty and it has a speed slower than theft, the running thread of the victim is interrupted and stolen. This approach allows adaptation to changes in processors speeds at runtime, but it dependents of how many times slower a processor would be to lose its thread, and the consequent migration of the thread probably adds costs.

NIEUWPOORT et al. (2006) proposed a variation of standard Work Stealing aiming to be efficient in a cluster of clusters with heterogeneous network. They provide the Satin (NIEUWPOORT; KIELMANN; BAL, 2000; NIEUWPOORT et al., 2006), which is a Java implementation of the Cilk model able to reach efficiency in distributed implementation of Java. Satin's Work Stealing has two-levels: first, the thief starts a non-blocking stealing attempt to a processor in a remote cluster; then, in parallel, it tries to steal tasks from processors in its local cluster. The first answered request is served. Usually, it is the local one, but the other request still get processed, and eventually the stealer will receive some tasks to process from a remote cluster, without having to wait idly for it. In simulations and experimentations, this Satin's Work Stealing was more efficient than a random choice of the victim in an environment with heterogeneous network connection.

As well as Satin, KAAPI (GAUTIER; BESSERON; PIGEON, 2007) also takes into account locality: several local steals are performed before to request remote ones. In KAAPI, each virtual processor has a set of threads and this set represent the deque. Virtual processors are assigned to physical processors (or cores in multi-core architectures). When a virtual processor becomes idle (when the executing thread blocks and suspends) the scheduling starts regarding if there is a ready thread waiting for the processor. If there is not, the KAAPI's scheduler chooses randomly a thread and verifies if it is ready to run. This rule is repeated until reach some thread able to be activated.

TBB aims at shared-memory architectures and follows the same principles of Cilk's Work Stealing: random choice of the victim, insertion and consumption from the bottom and stealing from the top of the ready deque (REINDERS, 2007). However, in TBB, the programs have a set of threads and each has a deque that stores non-computed TBB tasks, *i.e.* instances of the user-defined class that derives from the `task` abstract class.

3.4 Conclusion

This chapter exposed the main requirements to provide adaptive applications focused on runtime environment issues. The related works description was focused on three main aspects:

- **Interactions between RMS and adaptive applications:** implementation of the adaptive actions in parallel programs; the means used to exchange information between RMS and adaptive applications; and the specific RMS scheduling policies to deal with volatile processors;
- **APIs to develop adaptive applications:** description of the APIs structure that allows to deal with unpredictable variations in applications workload in distributed as well as in shared-memory contexts;

- **Scheduling issues taken into account the dynamicity of the parallel environments:** solutions required at starting time and to balance the workload at runtime.

This chapter concludes the first part of the thesis - Context: Adaptability in Parallel Environments - which aimed to contextualize the adaptability needs in current parallel environments. Furthermore, the impact of adaptability in parallel programming as well as in the runtime level was also presented. Table 3.5 summarizes the main aspects of adaptability (the adopted program structure; the runtime support; and Scheduling) in the related works. For space reasons, we used some acronyms: M/W to Master/Worker; LPar to Loop Parallel; F/J to Fork/Join; and WS to Work Stealing.

Table 3.5: Supporting adaptive applications: related works and their main features.

	Program Structure				**Runtime Support**		**Scheduling**
	SPMD	M/W	LPar	F/J	RMS	APIs	WS
Cilk			X*	X		X	X
OpenMP			X*	X		X	
TBB			X*	X		X	X
AMPI	X	X					
Leopold et al.		X					
Dynaco	X				X		
KAAPI				X		X	X
Satin				X			X
PCM	X	X			X		

* It appears in the literature, but it is not used for adaptability.

The second part of this thesis - Providing Adaptability to MPI Applications - will expose our efforts aiming to provide adaptive applications. Our work aims at distributed-memory environments such as clusters or cluster of clusters. In this context, the standard parallel API is the MPI. Thus, the question that guides our research is:

How to provide adaptability using MPI?

Part II

Providing Adaptability to MPI Applications

4 HOW TO PROVIDE ADAPTABILITY USING MPI?

This chapter intends to answer the question above. Therefore, it investigates how the MPI features can be used to provide an on-the-fly adaptation of the MPI applications. In this sense, firstly this chapter describes the dynamic process creation that is a feature figuring in MPI since 1998. Through this feature, MPI applications can spawn new MPI processes during their execution. The means and consequences of use dynamic process creation in MPI programs are described in Section 4.1.

We investigate two types of adaptability: *(i)* to volatile processors (*i.e.* processors with a dynamic availability); and *(ii)* to unpreditable needs (*i.e.* applications that adapt their execution to attend unpredictable needs). Thus, this chapter show as the MPI features can be used to implement adaptive applications. Section 4.2 describes the development of malleable MPI applications *i.e.* those who are able to deal with volatile processors. Likewise, Section 4.3 describes how to develop MPI applications following the explicit task parallelism, which adapt the extraction of the parallelism according to the architecture. To exemplify both, Section 4.4 presents a problem and its MPI implementations dealing with volatile processors and unpredictable needs. Finally, Section 4.5 concludes this chapter.

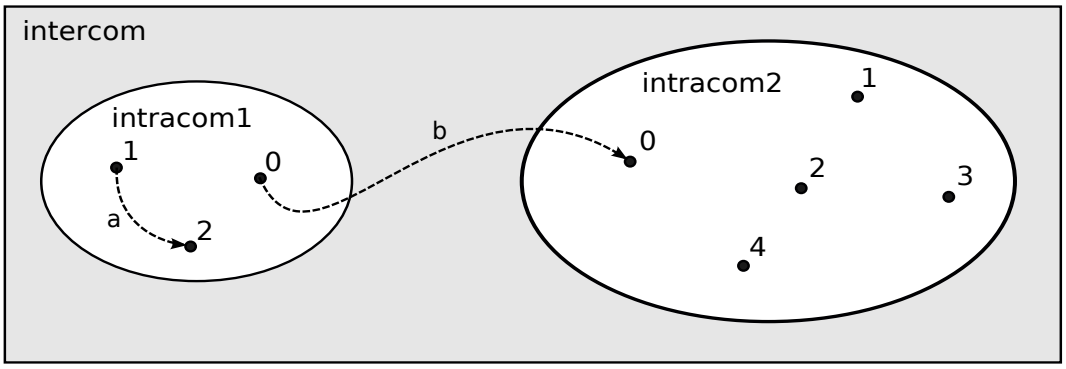
4.1 Using features of the MPI-2: Dynamic Process Creation

This section introduces the dynamic process creation in MPI programs. Firstly, Section 4.1.1 provides an overview of the MPI standard as well as the characteristics required to understand the remaining of this document. In the following, Section 4.1.2 introduces the `MPI_Comm_spawn` primitive and the main issues of its utilization. The communication relationships due to the dynamic process creation are described in Section 4.1.3. Concluding this introductory section, Section 4.1.4 presents an investigation about the overhead of dynamic processes creation in MPI programs performance.

4.1.1 Overview of the MPI Features

MPI (Message-Passing Interface) (GROPP; LUSK; SKJELLUM, 1994) is the standard parallel API for HPC in distributed-memory environments, which was devised by the MPI Forum[1]. It is an interface that allows inter-processes communications, in which are defined issues about point-to-point and collective communications, as well as organization of the programs such as groups and topologies of the processes, and communications contexts.

[1] *http://www.mpi-forum.org/* - last access in January 2011



a)
```
MPI_Send(&i, 1, MPI_INT, 2, tag, intracom1);
MPI_Recv(&i, 1, MPI_INT, 1, tag, intracom1, &st);
```

b)
```
MPI_Send(&j, 1, MPI_INT, 0, tag, intercom);      MPI_Recv(&j, 1, MPI_INT, 0, tag, intercom, &st);
```

Figure 4.1: Using intra and intercommunicators: *(a)* message exchanging in a local group using the intracommunicator `intracom1`; and *(b)* message exchanging between local and remote groups through the intercommunicator `intercom`.

An MPI program is composed by a set of processes. Each process has its unique identifier called *rank* that is defined at starting time, *i.e.* during the execution of the `MPI_Init` (every MPI source code starts by `MPI_Init` and concludes by `MPI_Finalize`). The universe of communication among MPI processes is defined by a structure called *communicator*. By default, every MPI process is inserted on a global communicator called `MPI_COMM_WORLD` (global communication universe), which provides point-to-point and collective communications.

MPI-1 (GROPP; LUSK; SKJELLUM, 1994) is the first MPI specification, in which programs follow strictly the SPMD program structure and all processes are created at starting time through the `mpirun` or `mpiexec` commands. These features attend the requirements of the many scientific problems, thus MPI becomes a popular API in scientific researches. However, there are problems that require some flexibility that cannot be achieved with MPI-1. Thus, the MPI Forum proposed the MPI-2 (GROPP; LUSK; THAKUR, 1999), which is an extension of the MPI-1, defining interfaces to create processes during the application execution, one-sided communications, parallel I/O, among other features. From the MPI-2 specification, programs can be developed as MPMD (*Multiple Programs Multiple Data*), *i.e.* multiple programs performing on multiple data.

After the diffusion of the MPI-2 norm, the concepts behind the communicators become essential to develop improved MPI programs. While MPI-1 programs, in general, use only the global communicator, MPI-2 programs can use several. Furthermore, there are two types of communicators:

- **Intracommunicator** allowing message exchanges among processes of the same local group. The global communicator `MPI_COMM_WORLD` is an intracomunicator;
- **Intercommunicators** allowing communications between processes of a local and remote groups.

To become clear these important concepts, Figure 4.1 exemplifies them. The message exchanging represented by the Send/Receive pair in Figure 4.1*(a)* happens in an intracommunicatior (`intracom1`): the process 1 sends an integer data to process 2, both participants of the same local group (dotted arrow *a* illustrates this communication). However, the Send/Receive Figure 4.1*(b)* happens in intercommunicator (`intercom`) that connects local and remote groups allowing the illustrated message exchange between both processes 0 (dotted arrow *b*). It is important to note that groups are named as local and remote according to whom is the sender (local) and who is the receiver (remote). Furthermore, the sender must know the rank of the receiver in the remote group.

4.1.2 Dynamic Process Creation

Although the MPI-2 was defined in 1998, its implementation only appears in the MPI distribution in 2005. The first distribution to provide dynamic process creation was LAM/MPI[2]. Nowadays, most distributions offer this feature, for instance, OpenMPI[3] (which is the continuation of the LAM/MPI project), MPICH2[4], etc.

MPI-2 defines two primitives to create processes at runtime: `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. The difference is that the first launches one or more images of the same binary, while the second can receive multiples binaries as parameter. We will focus on the first one. Conventionally, the process that calls `MPI_Comm_spawn` is named as **parent**, and the processes dynamically created are named as **children**. Figure 4.2 shows the parameters of `MPI_Comm_spawn`.

```
int MPI_Comm_spawn(
  char *command, ------------------> name of the binary to be spawned
  char *argv[], -------------------> arguments required by the binary
  int maxprocs, ----------> maximum number of processes to be started
  MPI_Info info, ----------------> information for the runtime system
  int root, --------------------------> rank of the parent processes
  MPI_Comm comm, -------------------> intracommunicator of the parent
  MPI_Comm *intercomm, -----> intercommunicator (parent and children)
  int array_of_errcodes[] ------------------------> array of errors
)
```

Figure 4.2: Parameters of the `MPI_Comm_spawn` primitive.

It is important to highlight some aspects of the `MPI_Comm_spawn` primitive. The fourth parameter, an `MPI_Info` variable, is a set of key-value pairs to inform the runtime system where and how to start the MPI processes. The sixth parameter is the intracommunicator of the parent, *i.e.* the communicator that represent the local group of the parent. This intracommunicator will be part of the intercommunicator between parent and children – the seventh parameter. Thus, parent and children can communicate as was shown in Figure 4.1. In the children side, the `MPI_Comm_get_parent` primitive allows to get the intercommunicator with the parent.

[2]LAM/MPI - *http://www.lam-mpi.org/* - last access in January 2011.
[3]OpenMPI - *http://www.open-mpi.org/* - last access in January 2011.
[4]MPICH2 - *http://www.mcs.anl.gov/research/projects/mpich2/* - last access in January 2011.

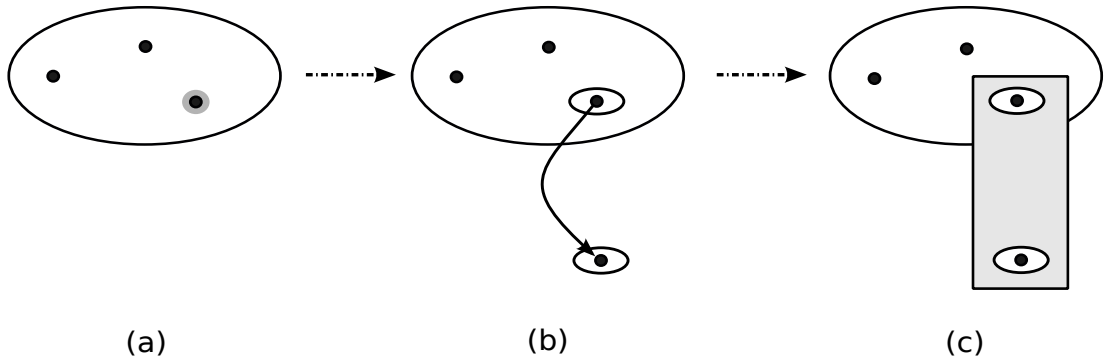


Figure 4.3: Dynamic process creation: *(a)* a process will call `MPI_Comm_spawn`, *(b)* a new process is being spawned in a remote group, and *(c)* the `MPI_Comm_spawn` returns an intercommunicator between parent and child.

Moreover, notice that communications between parent and children requires that both sides known the rank of the receiver in the remote group. The parent can easily know this information, since it is responsible to create and decide how many process will be created, thus it knows the rank of its children. However in children side, this is not so easy, because the local group of the parent may have many other processes, requiring the transmission of parent's rank to children. To avoid this transmission, another MPI predefined intracommunicator can be used: `MPI_COMM_SELF`. It is a self communicator of the MPI processes, *i.e.* it represents a local group composed uniquely by the caller process. Thus, any process, with any rank in the `MPI_COMM_WORLD`, has also its self intracommunicator in which it rank is always equal to 0. When `MPI_COMM_SELF` is used as the sixth parameter of the `MPI_Comm_spawn`, children always sent messages to parent using 0 as parent's rank.

Figure 4.3 illustrates the dynamic creation of an MPI process. In Figure 4.3*(a)* the marked process will call `MPI_Comm_spawn`. In Figure 4.3*(b)* the process is calling `MPI_Comm_spawn` using its `MPI_COMM_SELF` communicator (represented by the elipse around the process) and the new process is being spawned in a remote group. Finally in Figure 4.3*(c)* the `MPI_Comm_spawn` finishes and returns an intercommunicator between local and remote groups (illustrated as a rectangle).

Aiming to show the usage of `MPI_Comm_spawn` in practice, Figure 4.4 has an example of a dynamic MPI program[5] according to the Fork/Join program structure. In the left, there is the pseudo code of parent (`main.c`). It calls `MPI_Comm_spawn` (line 8) to create 8 images of the `task` binary - pseudo code in the right. Notice that as soon as children are started (execution of the `MPI_Init`– line 4), they call `MPI_Comm_get_parent` (line 5) to identify the intercommunicator with the parent, which is used to send back the results.

It is important to note that the same programming issues aiming efficient communications of traditional MPI program are also valid to dynamic ones. For instance, in Figure 4.4 the parent starts a non-blocking reception (`MPI_Irecv`– line 11) while it is performing multiple sends and after, it will wait for results from any children (`MPI_Waitany`– line 13). The adoption of these techniques can reduce the impact of synchronizations or barriers when they are required. Another possible improvement

[5]To provide a differentiation with static MPI programs, we often call programs able to spawn processes at runtime as dynamic MPI programs.

```
1. int main(int argc, char ** argv){
2.   << declarations >>
3.
4.   MPI_Init( argc, argv );
5.
6.   << computing the sequential work >>
7.
8.   MPI_Comm_spawn( "task", argv, 8, info, myrank,
                     MPI_COMM_SELF, &childrenComm, err );
9.   for (i=0; i<8; i++){
10.    MPI_Send( &in[i], size_in, datatype_in, i, 100,
                childrenComm );
11.    MPI_Irecv( &out[i], size_out, datatype_out,
                  MPI_ANY_SOURCE, 100, childrenComm,
                  request[i] );
12.  }
13.  MPI_Waitany( 8, request, j, status );
14.
15.  << continuing the execution >>
16.
17.  MPI_Finalize( );
18. }
```

pseudo code: main.c

```
1. int main(int argc, char ** argv){
2.   << declarations >>
3.
4.   MPI_Init( argc, argv );
5.   MPI_Comm_get_parent(parentComm);
6.   MPI_Recv( &in, size_in, datatype_in, 0,
               100, parentComm, status );
7.
8.
9.
10.
11.  << computing the input data >>
12.
13.
14.
15.
16.
17.  MPI_Send( &out, size_out, datatype_out,
               0, 100, parentComm );
18.  MPI_Finalize( );
19. }
```

pseudo code: task.c

Figure 4.4: Example of a dynamic MPI Program: a main process spawns 8 new tasks one as a Fork/Join program structure. In the left, there is the pseudo code of the parent (`main.c`), and in the right the pseudo code of the children (`task.c`).

is to use a collective communication instead of multiple `MPI_Send` calls to transmit the input data. Indeed, for the next version of the MPI standard – the MPI-3 – the MPI Forum is working on asynchronous primitive (`MPI_Icomm_spawn`), which tends to reduce the overhead of processes creation.

Another issue in dynamic MPI programs is that, in many cases, the parent must send input data to children. There are two possibilities: *(i)* input data is simple (*e.g.* one variable) and it can be set as children argument (second parameter of `MPI_Comm_spawn`); or *(ii)* the input data is complex (*e.g.* data structures) and it must be packed and sent using the intercommunicator (sixth parameter). In the pseudo code of Figure 4.4, the second option was implemented: after parent spawns children, it sends the input data (each child receives one part of the input data). As soon as children are started and get their intercommunicator with the parent, they receive the input. Notice that a data type called `datatype_in` was used (line 10 of `main.c` and 6 of `task.c`). This data type is a user-defined MPI data type (`MPI_Datatype`). Thus, the user data structures can be packed and sent using only one message instead of several. The creation of a data type involves many MPI primitives that were omitted, but can be easily found in MPI tutorials. Another data type is created to pack the output data from children to parent (`datatype_out` - line 17 of `task.c` and 11 of `main.c`).

Although there is a whole section to discuss about the communication topology of the dynamic MPI programs (Section 4.1.3), the simple example of Figure 4.4 allows identifying interesting aspects. Dynamic processes establish communications in a parent/children relationship through an intercommunicator. This means that dynamic MPI programs have a structured pattern of communication which was not usual in MPI programs, once, in general, MPI-1 programs have only one communication universe - `MPI_COMM_WORLD`. This feature can be used to solve problems that require more structured communications than a unique communicator. For instance, it can be used to implement fully strict (or well-structured) applications (BLUMOFE et al., 1996), as will be described in Section 4.3.

4.1.3 Communication Relationships among Dynamic MPI Processes

In the previous section, we have shown that dynamic process creation rises in a hierarchical data exchanging between parent and children processes – parent/children communication relationship. However, the MPI-2 has also defined how to establish communication among processes that do not share a communicator. In this case, two sets of processes that were launched independently (each with their own intracommunicator) may establish an intercommunicator in a client/server communication relationship.

MPI-2 defines client/server routines to connect servers (processes that indicate interest to accept connections) and clients (processes that connect to servers). Figure 4.5 illustrates these routines. In the server side, it opens a port to receive connections by `MPI_Open_port`. Then, the name of the service and the previously opened port are published through `MPI_Publish_name`. Afterwards, the server blocks in `MPI_Comm_accept` to wait for connections from clients. In the client side, it retrieves the pair (service name and port name) previously published by the server using `MPI_Lookup_name`. Then, `MPI_Comm_connect` establishes a connection with the server, *i.e.*, there is now an intercommunicator between server and client intracommunicators, allowing message exchanges.

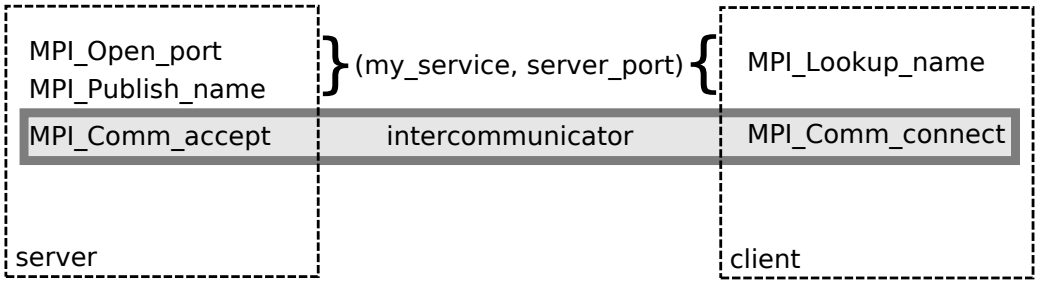


Figure 4.5: Steps to establish a client/server intercommunicator: *server* opens a port, publishes service and port name, and waits for connections; *client* lookups the service and port name, and connects with the server.

Figure 4.6 illustrates the MPI communicators and their communication relationships. In Figure 4.6*(a)*, there are five processes that share a unique universe of communication, *i.e.*, process can perform point-to-point or collective communications using a global intracommunicator such as `MPI_COMM_WORLD`. This is the communication pattern most used in MPI-1 programs. Figure 4.6*(b)* represents the parent/children communication relationship in which an intercommunicator allows communications parent and the dynamic MPI processes. In Figure 4.6*(c)*, there is a representation of the client/server communication relationship: an intercommunicator is established between two processes that do not share a universe of communication using the MPI routines introduced in the previous paragraph.

This section showed new communication relationships allowed by the MPI-2. On the one hand, these features increase the range of problems that can be solved using MPI. On the other hand, the programmers must dominate the features of these communication relationships and take them into account during the development of the MPI programs. Furthermore, when dynamic MPI processes have been used, there is an additional requirement aiming to control or manage their creation. For

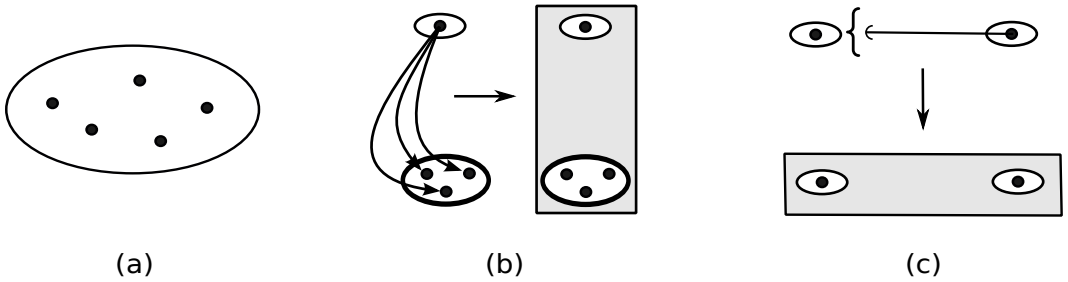


Figure 4.6: Illustration of the communication relationship: *(a)* global intracommunicator such as `MPI_COMM_WORLD`; *(b)* parent/children intercommunicator in a hierarchical communication relationship; and *(c)* client/server intercommunicator.

example, the programmer decides what binary, when (in what moment of the application execution), and where (in which processor) spawn through `MPI_Comm_spawn` parameters. In Sections 4.2 and 4.3, we will show uses of MPI-2 features as well as how the requirements of dynamic MPI processes can be answered.

4.1.4 Analysing the Overhead of Processes Spawning

A commonly question involving dynamic MPI processes is: What is the impact of spawn processes in application performance? This section attempts to answer this question through a comparison of the same problem solution implemented using dynamic against static processes. We took an MPI-1 implementation of a Master/Worker program and developed an MPI-2 version using `MPI_Comm_spawn` to dynamically create the workers.

The Master/Worker program chosen was the Mandelbrot Set, which is a classical embarrassing problem, used frequently to test parallel APIs (WILSON; IRVIN, 1995). It is a set of points in the complex plane that forms a fractal. Mathematically, Mandelbrot points are defined by the iteration of

$$\begin{cases} x^{'} = x^2 - y^2 + y_0 \\ y^{'} = 2xy + x_0 \end{cases} \tag{4.1}$$

to a given initial coordinates (x_0, y_0), until reaches an iteration limit or diverge the values. Thus, the computation of each Mandelbrot point is independent of the others.

The MPI-1 implementation of Mandelbrot has a master and as many workers as processors, all them created at starting time (*i.e.* the number of processes is set as an `mpirun` argument). The master starts by creating the bag-of-tasks. Each task aims the computation of a fractal block with 40×40 pixels, having as input the first (x,y) coordinates of the block and its size. Then, the master interacts with workers sending tasks and receiving matrix of fractal points on demand. Each worker receives a task, iterates the equations above, returns a matrix of points (*i.e.* the fractal block) and waits for another task. When the bag-of-tasks is empty the master sends a termination message to workers.

For the MPI-2, master and workers perform the same actions than the MPI-1, except that the master spawns the workers. We tested two alternatives: *(i)* workers compute one task and finalize, *i.e.* the master always spawn a worker when a task is pushed from the bag; *(ii)* workers compute a task and wait for others, *i.e.* the spawning of the worker happens only once (close to the MPI-1 design).

Figure 4.7 shows the execution time of Mandelbrot implementations with initial coordinates of $(0.5, 0.5)$ and an iteration limit of 30500. The number of workers increases until 24, 2 by 2, since the test infrastructure is a cluster with 12 dual processors nodes. It was used the LAM/MPI distribution to provide the dynamic process creation. Execution time values are averages of 15 executions with a standard deviation of at most 1.77. To clarify the difference among the application versions, Figure 4.8 presents the speedups. In both graphs, the solid line is the MPI-1 version and it has the best performance. The worst performance is when only one task is executed by each worker (MPI-2 v1 in graphs), thanks to the great number of spawns to compute fine-grain tasks. Furthermore, the performance gets worst as the number of workers increases, achieving a speed difference with MPI-1 of 4.2 to 24 workers. Finally, the MPI-2 version that only spawns workers once (MPI-2 v2 in graphs) achieves a low performance than MPI-1 but the speedup difference is around 2.2 to 24 workers, *i.e.* half of the other MPI-2 version.

Figures 4.7 and 4.8 confirm an expected behavior: to use dynamic processes to implement a fine-grain parallel application with a regular workload, aiming to a fixed number of processors is not efficient. On the other hand, a static MPI application such as the Mandelbrot using MPI-1, cannot deal with dynamicity as can the MPI-2 one. Figure 4.9 illustrates the execution of the MPI-2 Mandelbrot from 2 nodes of the cluster until 5, increasing 1 node each 45 seconds, in which the total number of processes is always equal to the number of available nodes.

The on-the-fly addition of the processors is provided by the LAM/MPI `lamgrow` command. In LAM/MPI each node of a cluster used by an MPI application runs a `lamd` daemon composing a network of daemons. Thus the `lamgrow` launches a `lamd` daemon in the further node and connect it with the other daemons in the network. When `lamgrow` concludes, the number of available nodes was increased. To identify increases in the number of available nodes, a procedure was inserted in the application source code. Thus, when there are new nodes, the application spawns one worker on each. The throughput in the Figure 4.9 shows that the application was able to adapt dynamically to use new nodes. As the number of nodes was been increased, there was an increasing in the number of tasks per second executed. Thus, the application was able to use nodes added at runtime as well as improves its performance.

Aiming to confirm the impact of use more nodes at runtime, Table 4.1 presents the efficiency of the application. The values in the table are based in a formulation proposed by HEYMANN et al. (2000), which takes into account the time spent by workers computing, and suspended waiting for computing, *i.e.*:

$$E = \frac{\sum_{i=1}^{n} T_{work,i}}{\sum_{n=1}^{n} T_{up,i} - \sum_{i=1}^{n} T_{susp,i}} \tag{4.2}$$

where,

- n is the number of workers;
- $T_{work,i}$ is the time that a worker i spent by making useful work;
- $T_{up,i}$ is the life time of the worker i;
- $T_{susp,i}$ is the time that a worker i keeps suspended.

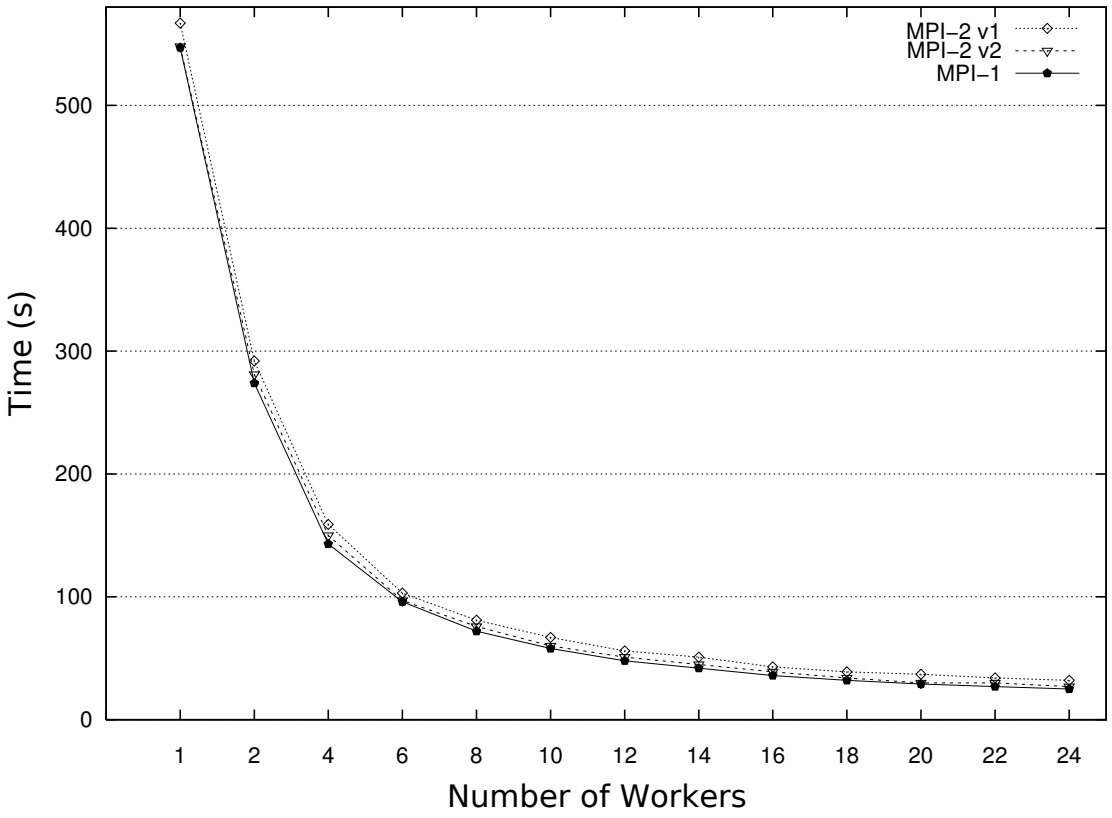


Figure 4.7: Timing *vs.* number of workers for Mandelbrot computation using MPI-1 (solid line) and two versions with MPI-2 (dotted lines).

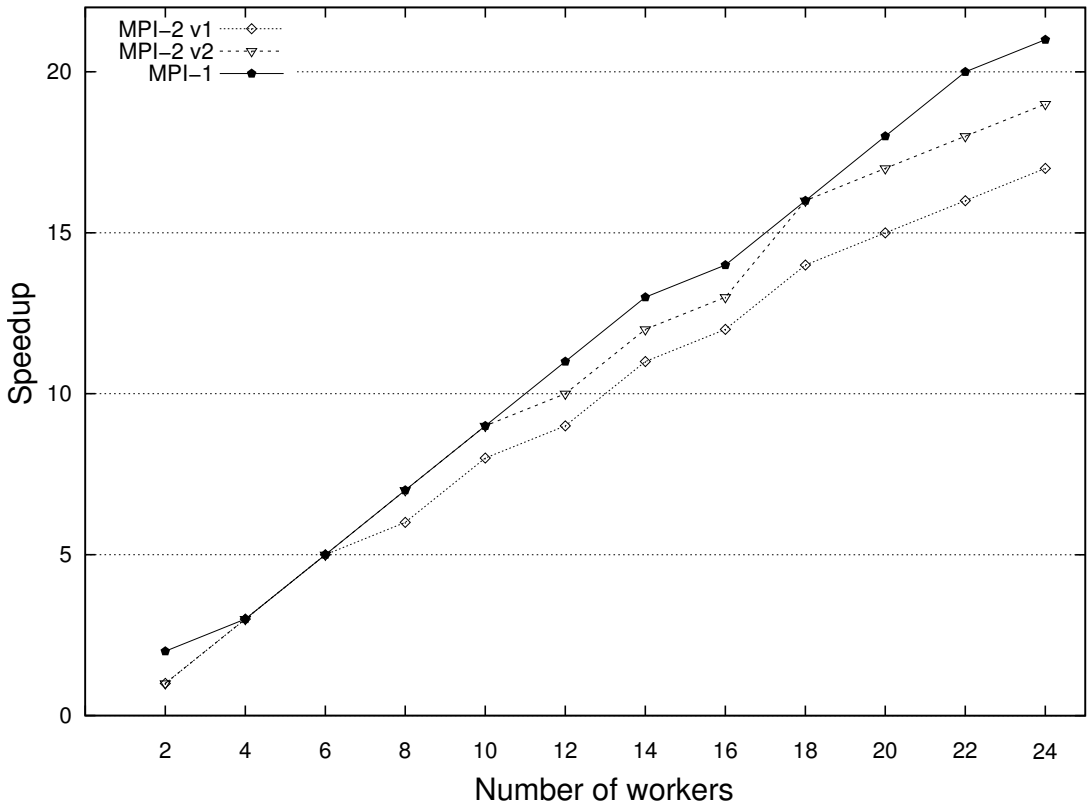


Figure 4.8: Speedups of the parallel implementation of the Mandelbrot Set using MPI-1 and MPI-2.

Table 4.1: Application efficiency while the number of workers increase.

Number of workers	Application Efficiency
2	1
3	0.99
4	1
5	0.97

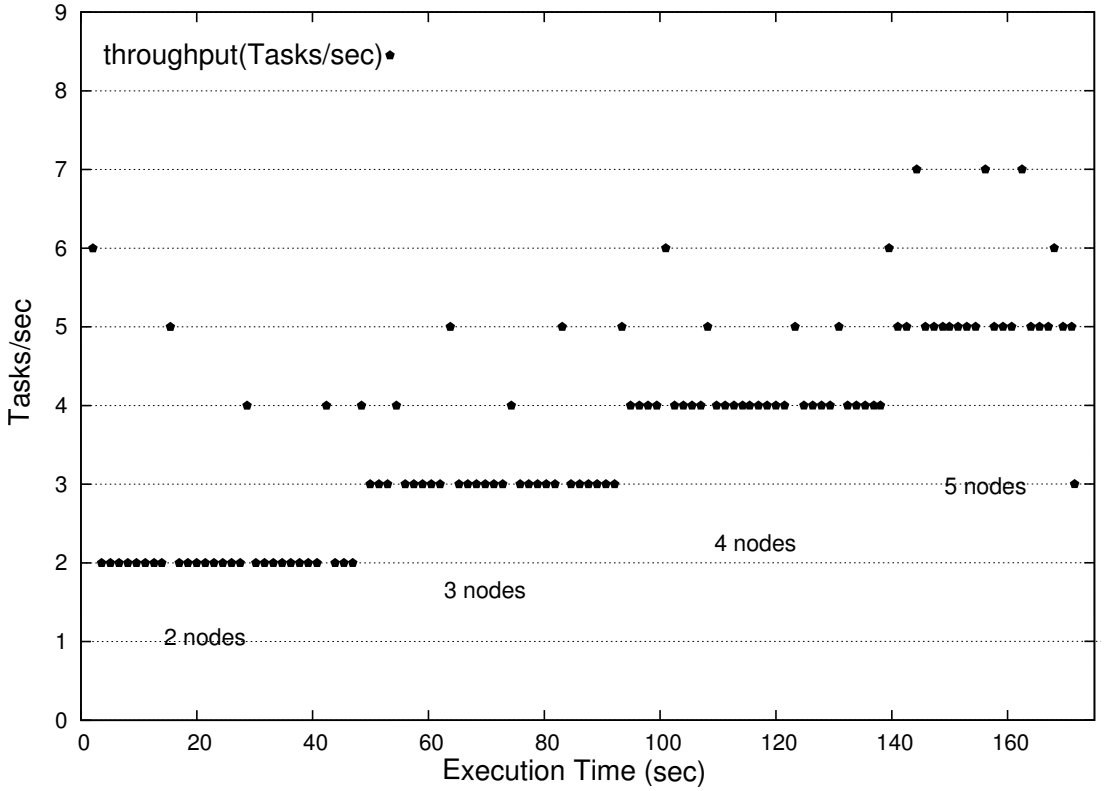


Figure 4.9: Throughput of MPI-2 Mandelbrot application (Tasks/second) while the number of processors was been increased.

As can be observed in Table 4.1, the efficiency of the application keeps closer to 1 while the number of workers increases, *i.e.* the application uses efficiently the further nodes added at runtime. Times required by the formulation were profiled during the application execution.

The experiments showed in this section were performed at the beginning of this thesis research. They verified the overhead of spawning processes at runtime. Furthermore, we perceived that this overhead can be acceptable in situations that require some flexibility from the MPI application, for instance, when the set of used processors varies at runtime. Thus, as a first step towards the execution of the MPI applications upon volatile processors, we tested here the use of `MPI_Comm_spawn` to adapt the application on-the-fly. We concluded that it can be efficient since we achieved improvements in the application throughput and efficiency. Based on this favorable scenario, we started to research other issues related to dynamicity in MPI applications, as will expose in Sections 4.2 and 4.3.

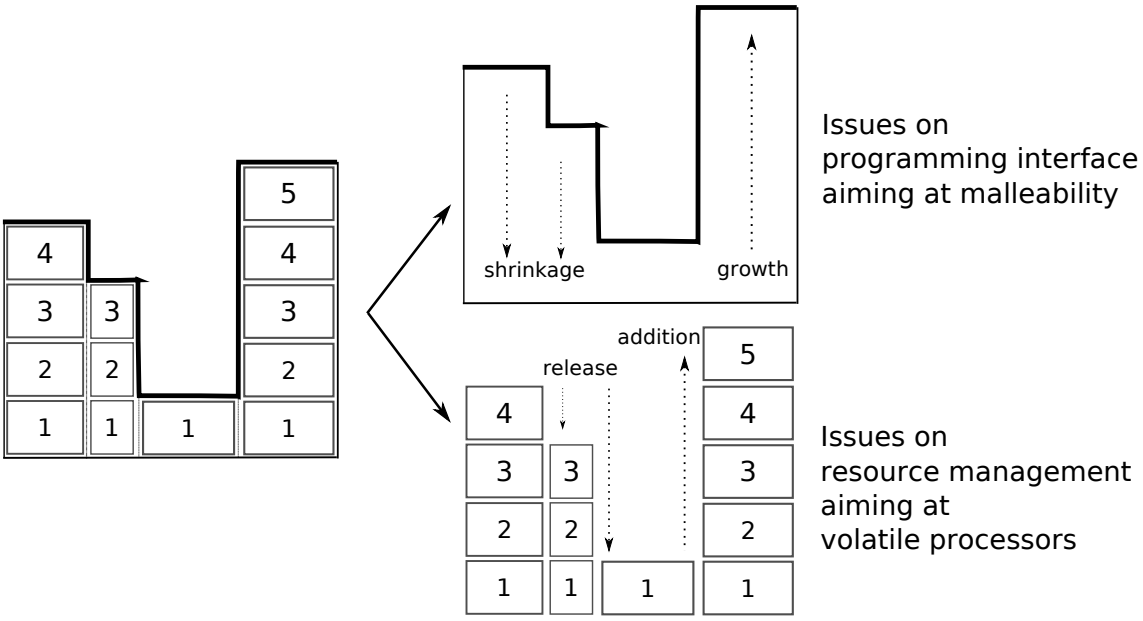


Figure 4.10: To run a malleable application it is required: *(i)* a programming level support aiming to provide a malleable behavior (growth and shrinkage at runtime); and *(ii)* a resource management able to deal with volatile processors (change the processors allotment at runtime).

4.2 MPI Applications dealing with Volatile Processors

Parallel applications able to execute upon a volatile amount of processors are known as Malleable, as was been introduced in Chapter 2. In addition, we showed that malleable applications require:

- Interactions with the RMS systems, aiming to exchange information about the availability of the processors;
- Procedures implementing the adaptive actions.

Figure 4.10 illustrates the twofold requirements of malleable application: on one hand the resource management must include policies able to deal with volatile processors; and in the other hand, the application must be able to grow and shrink at runtime. In Chapter 3 we shown how to the related works deal with these requirements. All these aspects of malleable application will be taken into account to answer the following question:

How to provide malleability in MPI applications?

The goal of this section is to show ways and means that the question above can be answered. Thus, Section 4.2.1 will describe the requirements of malleable MPI applications that must be supplied by the RMS, as well as when and what information must be exchanged between application and RMS. Further, Section 4.2.2 will expose the effect of dynamicity in the application development as the features of the adopted program structures.

4.2.1 RMS and Malleable MPI Applications Interactions

To illustrate the interactions between malleable MPI applications and RMS, we will analyse the three main moments of application execution:

- **At starting time** - the submission of the application including its launching on the allocated processors. When malleable applications are submitted, the users inform the range (minimum and maximum number) of required processors. To be able to run, a malleable application must have at least its minimum number of processors. For instance, a Master/Worker application requires at least one processor to start the master. When a malleable application starts, it uses all processors allocated by RMS (the decision is based in RMS policies such as will described in Section 5.1.2). Thus, a communication is required to transmit, from RMS to the malleable application, the number of allocated processors. Then, applications keep running statically until receive some changing notification (we name this notification as dynamic event);
- **When adaptive actions are required** - when occur changes in processors availability, either increasing (**growth action**) or decreasing (**shrinkage action**) the current number of processors. To require some change in a running malleable application, RMS sends a dynamic event that causes either a growth or a shrinkage action. This event, for instance, can be implemented as a signal or a message. Furthermore, applications must known the processors involved in adaptive actions. Thus, the RMS must inform, for example by a message, their number and identification. The application receives this information and it can either use the further processors or release the requested ones;
- **At ending time** - when the application finalizes. A malleable application ends similarly to a traditional MPI application, *i.e.* every MPI process (including those dynamically created) executes the `MPI_Finalize` primitive. Thus, the processors used by the application are released.

4.2.2 Developing Malleable MPI applications

Chapter 2 presented the adaptive actions that must be provided into parallel application for a runtime adaptation to volatile processors. Through these actions, the application is able to **grow** or **shrink** according to the processors availability.

The application growth involves: *(i)* a mechanism to identify increases in the set of available processors, *e.g.* a procedure to catch the signal sent by the RMS; and *(ii)* a procedure to allow the utilization of the new processors. In the malleable applications aimed in this work, this last point is implemented using dynamic process creation, then the growth procedure must ensure that the new MPI processes will be placed into the new processors.

The application shrinkage involves: *(i)* similarly in growth, a mechanism to identify that processors are being required by the RMS as well as which are them; and *(ii)* a procedure to release the required processors. Usually, there are two most used ways to release the processors: migrate the processes from the required processors; or finalize them (calling `MPI_Finalize`). We chose the second option since the migration of MPI processes is still an open issue and its investigation is out of the goals of this thesis.

However, shrinkage requires some further care since the MPI applications must continue to execute as expected even after losing some processes. Malleable applications in the context of this thesis implement the simplest solution: tasks that are being executed when the shrinkage starts will be restarted in future computations. Advanced solutions can restore the execution of the tasks, including a mechanism to save the data already computed avoiding their re-calculation. However, this solution brings extra transfer costs that depend of the problem being solved. We left the analysis of such a cost to future works.

The goal of this section was to provide an overview of the adaptive actions features aiming to allow MPI applications deal with volatile processors. However, implementation issues of growth and shrinkage actions are directly linked with the adopted program structure. Thus, we will focus on the most used program structures in MPI applications: Single Program Multiple Data - SPMD (Section 4.2.2.1); and Master/Worker (Section 4.2.2.2).

4.2.2.1 ***Malleability in SPMD MPI applications***

Similarly to Section 4.2.1, the requirement of the SPMD MPI applications to provide malleability will be described according to three execution moments:

At starting time. SPMD applications have a set of processes that are launched at starting time and then they will compute their own data. Notice that the global data is partitioned among processes at starting time. The distribution of the global data can happen either by sending chunks of data to be stored in the local memory, or by accessing shared data structures in which each process accesses its own subset of data. MPI collective operations can be used to improve data distribution. Usually, the number of processes of an SPMD MPI application is equal to the number of available processors. Malleable SPMD applications start following the same constraints that traditional MPI ones. Furthermore, they have their minimum number of processors equal to one, *i.e.* in the worst case, there is only one copy of the program computing the entire application data.

When are required adaptive actions. SPMD applications must have a procedure to allow the detection of changes in the availability of the processors. Figure 4.11 shows an example of such a procedure (`malleability_handler`). Moreover, the main issue about supporting malleability in SPMD applications is the **data redistribution**. Once each process has its own data set (determined at starting time), variations in the number of processes at runtime requires data transferences. Furthermore, the time spent by the adaptive actions will depend on how much data must be transfered.

Growth. Enabling an SPMD MPI application to use more processors than those available involves two issues: *(i)* to create new processes to run on new processors; and *(ii)* the redistribution of the application data in such a way that the new processes have their own data set to compute.
For MPI-2 malleable applications, the first issue is solved using the dynamic process creation (`MPI_Comm_spawn`). The second one involves the repartition of the data considering the updated number of processes. For instance, Dynaco uses a naive data redistribution which does not minimize and equilibrates the amount of data transfered in data redistribu-

```
/** malleability_handler **/

void malleability_handler(  )
{
  if( # processors increased )
    growth_action( nb_new );

  if( # processors decreased )
    shrinkage_action( nb_required );
}
```

```
/** growth_action **/

void growth_action ( int nb_new )
{
  if ( myrank ==  0 )
    MPI_Comm_spawn( "source_name", argv, nb_new,
                    info, root, MPI_COMM_SELF,
                    &comm_children, errcodes );
  num_procs += nb_new;
  redistribute_data( num_procs, myrank );
}
```

```
/** shrinkage_action **/

void shrinkage_action ( int nb_required )
{
  num_procs -= nb_required;
  redistribute_data( num_procs, myrank );

  if ( myrank IS_ONE_OF_THE releasing_ranks )
    MPI_Finalize(  );
}
```

Figure 4.11: Procedures aiming to support malleability in SPMD MPI programs: `malleability_handler` to detect changes in processors availability; `growth_action` to spawn processes into new processors and redistribute data; `shrinkage_action` to redistribute data and release the required processors.

tions. In consequence, the adaptive actions pay the costs of this naive redistribution and they can be not efficient in some cases. On the other hand, data redistribution is a open research field and there are initiatives offering efficient libraries (RAUBER; RUNGER, 2005; ZHANG et al., 2009). These libraries can be used in the context of the malleable applications to reduce the costs of data transfers. Figure 4.11 shows a pseudo code for growth actions (`growth_action`): one of the SPMD processes (the process 0 in our example) spawns processes into the new processors (set on `info`), then all processes update the current number of processors and make a new data redistribution. Notice that data redistributions are collective operations that use the MPI communicators to transfer data.

Shrinkage. The releasing of processors by SPMD applications requires: *(i)* to transfer data from processes that will be stopped, *i.e.*, the processes running upon the required processors and *(ii)* safely finalize them.
The first issue can be solved using the same solution that in growth case, where data redistribution libraries can improve the data transferring. The second issue requires a procedure able to identify that a specific processor is required and, after the completion of the data transferring, finalizes it, *i.e.* to call `MPI_Finalize`. Figure 4.11 shows a pseudo code for shrinkage actions (`shrinkage_action`): SPMD processes update the total number of processors, call `redistribute_data` to transfer data from processes in releasing processors and then, each process test if it must finalize or not. Notice that this procedure requires means to identify which are the releasing processors.

At ending time. SPMD applications finalization includes the recombination of data when they are stored in local memory. Again, MPI collective operations can be used to improve the transfers. After that results are merged, the computation is shut down through `MPI_Finalize`.

Section 4.4 will present deepest details of the adaptive actions showed here, once there we will implement a problem solution according to the requirements of malleable SPMD MPI applications.

4.2.2.2 ***Malleability in Master/Worker MPI applications***

This section aims to show the requirements of malleable Master/Worker applications, as well as how the MPI-2 features can aid to supply them.

At starting time. In a malleable Master/Worker MPI application, at starting time there is only the master running. It initiates the computation setting up the problem and creating the bag-of-tasks. According to the different ways that a Master/Worker application can be implemented using MPI, now the master can:

1. To launch workers using the `MPI_Comm_spawn`;
2. To accept connections from workers that have been launched by independent `mpirun/mpiexec` in a client/server relationship (`MPI_Comm_connect`/ `MPI_Comm_accept`);
3. To perform as an SPMD program – master and workers have the same source code and are launched by the same `mpirun/mpiexec`, being the behavior of each process determined as their rank (this is often found in MPI-1 Master/Worker applications).

We use the first option in our malleable MPI applications. Once workers are initiated, they receive tasks from the bag and start to compute. A malleable Master/Worker application has the same initial requirements of the static one: one worker will execute on each available processor. In addition, the minimum number of required processors is one, which enables the starting of the master (setting up the problem and creating the bag of tasks).

When are required adaptive actions. As well as in SPMD programs, it is required a procedure to allow the detection of changes in processors availability as showed in Figure 4.11 - `malleability_handler`. Furthermore, the main issue of malleability in Master/Worker programs is the **management of the workers**: dynamic creation and finalization.

Growth. When there are new processors available, a Master/Worker program easily uses them by launching new workers on them. According to the classic algorithm, once workers are created, they receive tasks and start to compute. Thus, the Master/Worker program structure takes care of forecasts some load to the new processors.

Using dynamic process creation, when growth is required, the master creates new workers and the intercommunicator enables them to receive tasks from the master bag of tasks. For example, PCM migrate processes (workers) to new processors, using `MPI_Comm_spawn` to implement migration: the migrating process spawns a child in the new processor and sends its current state (MAGHRAOUI; SZYMANSKI; VARELA, 2006). Thus, the migrated processes are automatically connected with the MPI application. Furthermore, PCM is able to split running processes before migrations to have enough processes to migrate and divide the workload in such a way to have a balanced data distribution. This last issue is

```
/** growth_action **/

void growth_action ( int nb_new )
{
  int i;
  for( i = 0; i < nb_new; i++ )
  {
    MPI_Comm_spawn( "worker", argv, 1, info, myrank, MPI_COMM_SELF,
                    &w_comm[curr_w], err );
    MPI_Send( &bag[next_t], 1, task, myrank, TAG, w_comm[curr_w] );
    MPI_Irecv( &res[curr_w], 1, result, MPI_ANY_SOURCE, MPI_ANY_TAG,
               w_comm[curr_w], &reqs[curr_w] );
    curr_w++;
    next_t++;
  }
}
```

```
/** shrinkage_action in Master **/

void shrinkage_action ( int nb_required )
{
  int i, j;
  num_procs -= nb_required;
  for ( i = 0; i < nb_required; i++ )
  {
    j = get_releasing_rank( i );
    MPI_Recv( &task, 1, task, MPI_ANY_SOURCE,
          MPI_ANY_TAG, w_comm[ j ], &st );
    push_task_in_bag( task );
  }
}
```

```
/** releasing in Worker **/

void releasing( )
{
  task = getCurrentTask( );
  MPI_Send( &task, 1, task, myrank,
            TAG, m_comm );
  MPI_Finalize( );
}
```

Figure 4.12: Procedures aiming to support malleability in Master/Worker MPI programs: `growth_action` to spawn processes into new processors sending tasks to them; `shrinkage_action` to allow the Master to identify the workers that must be stopped and receive their tasks; `releasing` to allow the worker to get the current task and send it back to the master before finalizing.

related to migration and can be left aside in our proposal that creates further workers to use new processors. Figure 4.12 illustrates a possible growth action to Master/Worker programs: to each new processor, the master spawns a worker, which will execute the next task from the bag. The current number of tasks and workers is updated.

Shrinkage. When a Master/Worker program must release some processors, processes running on them must be stopped in such a way do not compromise the application execution. The restriction of this program structure is that the master cannot be stopped, since it is responsible for the computation management. On the other hand, all processors that have workers are candidates to be released.

To avoid the application crash during shrinkage, workers must be properly finalized and their tasks must be replaced into the master bag of tasks. Thus, these tasks will be eventually computed in the future, maintaining the correctness of the application results. Furthermore, the master must knows which workers had been finalized to avoid dealocks waiting for finalized workers. Figure 4.12 shows the two required procedures: `shrinkage_action` in master side to receive tasks from workers in releasing processors and push they to bag of tasks; `releasing` in worker side to get the current task and send it back to master before finalizing.

At ending time. In the classic Master/Worker algorithm, when the bag of tasks is empty, the workers are informed and finalize (`MPI_Finalize`). The master collects the workers results, merges them when necessary, and shuts down. Moreover, since the number of workers had been updated during the adaptive actions, a malleable Master/Worker application ends as expected.

In the last two sections, there were introduced the main requirements to provide malleability in MPI application, according to features of the SPMD and Master/-Worker program structures. In addition, we proposed solution to support these requirements taking into account the MPI-2 features. Section 4.4 will exemplify the development of malleable MPI applications in practice: for a target problem, malleable SPMD and Master/Worker programs will be provided.

4.3 MPI Applications dealing with Unpredictable Needs

Applications able to adapt to unpredictable needs are named evolving as Feitelson and Rudolph job classification (Chapter 2), since they only known about the needs at runtime. Furthermore, Chapter 3 introduced the Explicit Task Parallelism and its increasing use in current researches as an efficient way to extract the degree of parallelism required by the target architecture. We consider that applications implemented according to the explicit task parallelism can be seen as evolving. Thus, this section focuses on the development of these applications with MPI as well as their main requirements. In the following, Section 4.3.1 describes how it is possible to implement MPI applications as the explicit task parallelism paradigm, and Section 4.3.2 shows the requirements of these applications.

4.3.1 Developing Explicit Task Parallelism in MPI: D&C Algorithms

According to previously described in Section 2.5, Fork/Join is the program structure most appropriate to deal with unpredictable changes in the application workload. As an example, we showed that Divide and Conquer (D&C) algorithms can be implemented following the Fork/Join program structure (Section 2.5.4). Furthermore, the explicit task parallelism paradigm fits with D&C algorithms: the divide phase creates new tasks (dynamically unfolding the parallelism) until the problem become trivial to be solved sequentially. Notice that the potential parallelism is limited by the size of the problem, *i.e.* the input data.

This section aims to show the development of D&C applications using dynamic process creation to unfold the parallelism at runtime. As an illustration, the pseudo code of a D&C application provided in Figure 2.7 was translated to an MPI application, as can be seen in Figure 4.13. Our previous work described issues in development of D&C MPI applications (PEZZI et al., 2006).

Figure 4.13 exposes two pseudo codes: `main.c` and `divide_and_conquer.c`. The first starts the D&C computation setting up the problem (represented by `getProblem` function - line 5) and spawning the first divide_and_conquer process (line 7). Furthermore, the main process sends the problem (line 8) and blocks waiting for the child results (line 9), *i.e.* the problem solution. The `divide_and_conquer.c` pseudo code starts by getting the parent intercommunicator (line 5), since divide_and_conquer processes are always dynamically created. Through this intercommunicator, they receive the problem, *i.e.* the input data - line 6.

```
   /********    main.c    ********\
```

```
 1.int main(int argc, char ** argv)
 2.{
 3.  << Declarations >>
 4.
 5.  problem = getProblem();
 6.
 7.  MPI_Comm_spawn( "divide_and_conquer", argv, 1, info, myrank,
                     MPI_COMM_SELF, &child_comm, err );
 8.  MPI_Send( &problem, size_prob, datatype_prob, 0, tag, child_comm );
 9.  MPI_Recv( &solution, size_sol, datatype_sol, 0, tag, child_comm, st );
10.  ...
11.}
```

```
   /********    divide_and_conquer.c    ********\
```

```
 1.int main(int argc, char ** argv)
 2.{
 3.  << Declarations >>
 4.
 5.  MPI_Comm_get_parent(parent_comm);
 6.  MPI_Recv( &problem, size_prob, datatype_prob, 0, tag, parent_comm, st );
 7.
 8.  if( problem is trivial )
 9.  {
10.    solution = compute( problem );
11.  }
12.  else
13.  {
14.    subProblem = split( problem );
15.    for( i = 0; i < 2; i++)
16.    {
17.      MPI_Comm_spawn( "divide_and_conquer", argv, 1, info, myrank,
                         MPI_COMM_SELF, &children_comm[i], err );
18.      MPI_Send( &subProblem[i], size_prob, datatype_prob, i, tag,
                   children_comm[i] );
19.      MPI_Irecv( &subSolution[i], size_sol, datatype_sol, i, tag,
                    children_comm[i], req[i] );
20.    }
21.    MPI_Waitany( 2, req, j, st);
22.
23.    solution = conquer( subSolution );
24.  }
25.  MPI_Send( &solution, size_sol, datatype_sol, 0, tag, parent_comm );
26.  ...
27.}
```

Figure 4.13: Pseudo code of a D&C MPI application: `main.c` gets the problem and spawns one `divide_and_conquer` process; `divide_and_conquer.c` computes the problem if it is trivial sending the result to its parent, otherwise divides the problem in two parts, spawns two children to compute them, waits for children results, and sends the merge of sub-problems solutions to the parent.

Notice that to perform this first receive, as well as the other message exchanges, the processes must know the size of the problem (second parameter of `MPI_Recv`- line 6). As it can only be known at runtime (the size depends of the how many divisions have been made), the parent process must inform to children such a value either as an argument of the new processes (second parameter of `MPI_Comm_spawn`- line 7 of `main.c`) or by a further message sent before the receiving of the problem (*i.e.* data transfers use two messages: one to sent the size of the problem, and another to transfer the problem data). In our example, we omitted it but the size is set as process argument. Furthermore, this example takes advantage of the `MPI_Datatype` to pack data in message exchanges (third parameter in `MPI_Send`, `MPI_Recv`, and `MPI_Irecv`).

When the problem is available, divide_and_conquer processes test if it can be trivially solved (line 8). When it can, the problem is computed sequentially and the solution is sent back to the parent - lines 10 and 25. Otherwise, the processes enter in the divide phase splitting the problem into sub-problems (line 14). Two new divide_and_conquer processes are created to deal with each sub-problem (`for` of lines 15 t0 20) until the problem becomes trivial. The conquer phase starts when the result are returned by the children (`MPI_Waitany`- line 21), in which the process merges them in the `conquer` function - line 23. After that, the solution is sent to the parent - line 25.

Observing the processes creation in D&C MPI pseudo code of Figure 4.13, it is possible verify their structured dependency according to the fully strict model (like Cilk programs). Parent processes can only continue their execution when children satisfy the dependency among them – sent the results. Thus, the divide phase creates the dependencies: new processes are spawned and the parents wait for them. On the other hand, in conquer phase dependencies are solved: processes receive children results, merge and sent them to parent before finalizing.

4.3.2 Requirements of Explicit Tasks MPI Applications

To develop explicit task programs, it is required to define: *(i)* what is a task; *(ii)* what are the dependencies among tasks; and *(iii)* to schedule them. In this section we will analyse these issues taking into account the MPI-2 features.

The MPI specification defines MPI tasks as having their own address space and most MPI distributions map a task to an OS process. In the pseudo code of Figure 4.13, this statement is followed: one new process is spawned to each task generated at runtime. However, this decision brings an important issue: how to reach an efficient granularity to dynamically generated tasks?

In D&C applications, the granularity change at runtime. Basically, the three moments of the application execution impact on workload:

- **Dividing:** granularity decreases as problem is split into sub-problems. Notice that it can involve large amounts of data;
- **Computation:** once tasks have a computable size causing the stop of dividing phase, they are sequentially computed;
- **Conquering:** granularity increases as sub-problems solutions are merged. Again, it can involve large amounts of data.

In general, it is expected that computation is the dominant part of the execution. Thus, to implement efficient D&C MPI application, the stop condition of the divide phase must be set to provide sufficient workload to overlap the costs of data transmissions. Moreover, as we intend to create new processes at runtime, the computation must also overlap the costs of spawn them.

The straightforward way to solve dependencies among dynamic MPI tasks is through message exchanging as showed in Figure 4.13. The intercommunicator between parent and children helps to solve their structured dependency (as the fully strict model). On the one hand, the parent blocks waiting for children results (`MPI_Recv` in `main.c` and `MPI_Waitany` in `divide_and_conquer.c`). On the other hand, the children always solve their dependency with the parent before finalizing (last `MPI_Send` in `divide_and_conquer.c`).

Issues related to scheduling dynamic MPI tasks can be seen in two topics: *(i)* their mapping into the physical processors; and *(ii)* to balance the workload among them. For the first, the MPI standard does not specify mapping schemes, but each MPI distribution provides the allocation of dynamic processes between the available processors. In general, the default mapping is simple as the Round-Robin strategy. When the LAM/MPI distribution began to support `MPI_Comm_spawn`, its Round-Robin mapping was unable to place the dynamic processes in the expected processor. In CERA et al. (2006) we described and proposed a solution to this problem. Some current MPI distributions, such as the OpenMPI, already solved this kind of problem offering a satisfactory distribution of the dynamic MPI processes, helping the exploration of explicit task parallelism.

For the second topic, the most used strategy to provide load balancing among dynamically generated tasks is the Work Stealing (Section 3.3.2). In PEZZI et al. (2007) we studied the adaptation of this strategy on MPI programs with dynamic processes, in which it was implemented inside of the application source code. Through this work, we identified that Work Stealing can be used in MPI applications since some constraints are taken into account. For instance, the impact of the higher cost of steals in distributed-memory environments than in shared-memory ones, and the hierarchical structure of the communication channels.

This section highlighted issues and challenges to provide efficient implementations of explicit task parallelism in MPI. A further analysis of these issues will be provided in Chapter 6. However, the next section shows how the explicit task parallelism can be used to solve a problem taking advantage of the dynamic processes creation.

4.4 Exemplifying the development of Adaptive MPI Applications

Aiming to illustrate the design of the adaptive MPI applications as described in Sections 4.2 and 4.3, this section implements matrix multiplication provide malleable and evolving behaviors. The problem is solved computing the multiplication of two matrices A and B with $n \times n$ elements, storing the result in a matrix C also with $n \times n$ elements:

$$\begin{pmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nn} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1n} \\ B_{21} & B_{22} & \dots & B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & \dots & B_{nn} \end{pmatrix}$$

$$C_{n\times n} = A_{n\times n} \times B_{n\times n}$$

Following parallel implementations of the matrix multiplication decompose the matrices into $k \times k$ blocks $M_{i,j}$ (i,$j = 1 \dots k$), in which the size of each block is $\frac{n}{k} \times \frac{n}{k}$. Through this decomposition, $\forall i$,$j = 1 \dots k$ there is a product of two matrices with $k \times k$ elements as:

$$C_{ij} = \sum_{k=1}^{k} A_{ik} \times B_{kj}$$

Considering p the number of available processors and $p < n$, the value of k can be chosen between 1 and $\sqrt{p}$. Thus, for $k == \sqrt{p}$ there is exactly 1 block per processor, and for $k < \sqrt{p}$ there are more blocks (and tasks) than processors.

4.4.1 Examples of Malleable MPI Applications

This section presents how to the matrix multiplication can be implemented with SPMD (Section 4.4.1.1) and Master/Worker (Section 4.4.1.2) program structures to deal with volatile processors.

4.4.1.1 ***SPMD Example***

Being N the number of processes and p the number of processors, the SPMD matrix multiplication starts with $N == p$, *i.e.* there are as many processes as available processors. Listing 4.1 shows the `matmul-spmd.c` source code of an MPI matrix multiplication. The initialization happens in lines 9 to 12, in which the MPI starts and each process gets its unique identifier (`rank`), the total number of processes (`numprocs`), and the size of input matrices (`n`). Matrices $A_{n\times n}$ and $B_{n\times n}$ are partitioned into blocks (sub-matrices) A_{ij} and B_{ij} having $\frac{n}{k} \times \frac{n}{k}$ elements each and $k == \sqrt{p}$. In other words, the input data is partitioned in such a way that all units of processing have their own set of data. If there is only the minimum number of processors available, p and k are 1, resulting in a block with $\frac{n}{1} \times \frac{n}{1}$ elements or the entire matrices computed by one process.

The partitioning of the data happens in lines 15 to 20: the size of sub-matrices is calculated according to the number of processes and it is stored in the variable `K`; then, sub-matrices are allocated in the processes local memory; and each process gets its data as its rank and the `K` value. Notice that in `get_input_*` implementation, data can either be received from other processes or be read from files. Processes start to compute their own data in lines 23 to 29. No interactions are required during this phase once the computations depends only the sub-matrices data. Afterward, processes gather the output sub-matrices (line 31) to compose the final result and then they shut down (`MPI_Finalize` in line 33).

Listing 4.1: Source code of SPMD matrix multiplication using MPI: `matmul-spmd.c`.

```
1  int main (int argc, char **argv){
2    int n;                          /* order of the matrices */
3    int myrank;                     /* unique identifier of the processes */
4    int numprocs;                   /* number of processes */
5    int K;                          /* order of the submatrices */
6    int i, j, k;
7    double *sub_A, *sub_B, *sub_C;
8
9    MPI_Init(&argc, &argv);
10   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
11   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
12   n = atoi(argv[1]);               /* getting n value from the arguments */
13
14   /* data partitioning */
15   K = (n / numprocs);             /* determining the submatrices order */
16   sub_A = (double *)malloc(sizeof(double) * K * K);
17   sub_B = (double *)malloc(sizeof(double) * K * K);
18   sub_C = (double *)malloc(sizeof(double) * K * K);
19   get_input_A(sub_A, myrank, K); /* getting input data */
20   get_input_B(sub_B, myrank, K);
21
22   /* data computation */
23   for(i = 0; i < K ; i++){
24     for (j = 0; j < K; j++){
25       for (k = 0; k < K ; k++){
26         sub_C[i * K + j] += sub_A[i * K + k] * sub_B[k * K + j];
27       }
28     }
29   }
30   /* gathering results */
31   gather_output(sub_C, myrank, K);
32   /* ending the computation */
33   MPI_Finalize();
34 }
```

Allowing the source code in Listing 4.1 to deal with volatile processors, it must have a procedure to test the availability of the processors. Such a procedure was been shown in Figure 4.11 - `malleability_handler`. For instance, in Listing 4.1, the procedure can be inserted inside of the third nested loop (lines 25 to 27) causing its iterative call. However, in this approach the time between one call and another will be limited by the time spent to compute one iteration, and in some cases it can be long. Another approach is to launch a thread to execute the handler procedure, allowing an immediate response to changes.

These responses to changes are the application growth allowing using more processors than currently, and the application shrinkage allowing releasing some processors. Figure 4.11 illustrated how these procedures can be implemented to SPMD programs. When the application growth, it must:

- **To create new processes** in further processors to maintain $N == p$. Using `MPI_Comm_spawn`, one of the running processes can spawn the required processes. However, an additional care must be take with the communications relationships. As all SPMD processes are launched together by `mpirun/mpiexec` they communicate using the global communicator (`MPI_COMM_WORLD`). When processes are spawned, there is an intercommunicator connecting the parent and children communicators. Thus, only the parent can exchange message with the children;
- **To update the number of processes** to add the new ones;

- **To redistribute data.** The new processes must receive some data to compute, thus the size of the sub-matrices must be recalculated to keep $k == \sqrt{p}$. This will cause an updating in all sub-matrices. Data transfer libraries can be used to improve these operations. Furthermore, the communications relationship among processes impact in data redistribution, mainly with the new processes that are exclusively accessed through their parent.

On the other hand, the shrinkage of an SMPD MPI application requires:

- **To update the number of processes** to decrease the releasing ones;
- **To redistribute data.** The same issues of growth are valid to shrinkage: recalculate the size of sub-matrices and transfer data taking care of the communication relationships;
- **To finalize processes** that are placed in the releasing processors.

Data redistributions must also take into account the already computed data to avoid re-computations. For instance, some dynamic programming method can be adopted to express the state of the computation. Specifically in matrix multiplication, an auxiliary structure, such as an array, can be set according to the elements of the sub-matrices `sub_C` are being computed. Thus, in some cases, the input data related to already computed data may be left aside in redistributions, reducing the amount of data to be transferred.

As could be observed in this section, to support volatile processors in SPMD applications is not trivial. Most part of difficulties rise from data redistributions and the communication relationship among processes. These issues can be solved during the program development, but requires extra efforts from programmers or an extra support from the programming environment. For example, PCM (Section 3.1.1) provides data distributions taking advantage of checkpoints and a database to store the application data, which can be accessed by the processes using a key with their rank. Furthermore, to avoid problems in communication among dynamic and static processes, PCM implements its own communicator, the `PCM_COMM_WORLD` since the global communicator of MPI do not support changes in the number of processes as well as in their ranks at runtime.

4.4.1.2 ***Master/Worker Example***

Listing 4.2 shows the source code of the master: `master.c`. In lines 14 to 17, the MPI process starts up, gets its rank and its arguments variables: `n` - size of the matrices; and `p` - number of available processors at starting time. In lines 20 to 23, the matrices `A`, `B` and `C` are allocated and filled up (`get_input`). The bag of tasks is created by `create_bag` (line 25), which divide the input matrices into sub-matrices A_{ij} and B_{ij} having $\frac{n}{k} \times \frac{n}{k}$ elements each, being $k < \sqrt{p}$. Thus, there are more tasks (sub-matrices) in the bag than units of processing (`p`).

The structure `s_task_t` stores the tasks input, *i.e.* the coordinates (`i` and `j`), the size and elements of the sub-matrices. The results of multiplications are stored into `s_result_t` structure that has the coordinates, the size and the elements of the resulting sub-matrix (`sub_C`). Both structures are used in master and worker sides.

Master launches p workers using `MPI_Comm_spawn`, sends one task to each worker and calls an asynchronous receive to wait for their results (lines 30 to 36). Notice that user-defined `MPI_Datatype task_t` and `result_t` are used to encapsulate `s_task_t`

Listing 4.2: Pseudo code for a Master/Worker matrix multiplication: `master.c`.

```
1  int main(int argc, char **argv) {
2    int myrank, i, w, finished = 0;
3    int n;                          /* order of the matrices */
4    int p;                          /* number of processors */
5    double *A, *B, *C;
6    s_task_t *bag;                  /* bag-of-tasks */
7    s_result_t *res;                /* results */
8    MPI_Comm w_comm[MAX];           /* intercomm between master and workers*/
9    MPI_Status st;
10   MPI_Info info;
11   MPI_Request reqs[MAX];
12   MPI_Datatyepe task_t, result_t;
13
14   MPI_Init(&argc, &argv);
15   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
16   n = atoi(argv[1]);
17   p = atoi(argv[2]);
18
19   /* allocating the matrices */
20   A = (double *) malloc(sizeof(double) * n * n);
21   B = (double *) malloc(sizeof(double) * n * n);
22   C = (double *) malloc(sizeof(double) * n * n);
23   get_input(A, B, n);    /* getting input data */
24
25   bag_size = create_bag(A, B, n, bag);    /* creating the bag of tasks */
26   allocate_results(res, p);               /* allocating results */
27   task_t = create_task_Datatype();
28   result_t = create_result_Datatype();
29
30   for (i = 0; i < p; i++) {    /* launching workers */
31     MPI_Comm_spawn("worker", MPI_ARGV_NULL, 1, info, myrank, MPI_COMM_SELF,
32                    &w_comm[i], errcodes);
33     MPI_Send(&bag[i], 1, task_t, myrank, TAG, w_comm[i]);
34     MPI_Irecv(&res[i], 1, result_t, MPI_ANY_SOURCE, MPI_ANY_TAG, w_comm[i],
35               &reqs[i]);
36   }
37   while (p - finished > 0){
38     MPI_Waitany(p, reqs, &w, &st);
39     store_result(res[w]);
40     if(i < bag_size){
41       MPI_Send(&bag[i], 1, task_t, myrank, TAG, w_comm[w]);
42       MPI_Irecv(&res[w], 1, result_t, MPI_ANY_SOURCE, MPI_ANY_TAG, w_comm[w],
43                 &reqs[w]);
44       i++;
45     }else{
46       MPI_Send(&task_empty, 1, task_t, myrank, TAG, w_comm[w]);
47       finished++;
48     }
49   }
50   MPI_Finalize();    /* ending the computation */
51 }
```

and `s_result_t` structures in messages exchanges (these types are created in lines 27 and 28). Then, the master waits for any result while there are alive workers – lines 37 to 49. When a worker sends its results, the master stores them and sends another task to the worker while there are tasks in the bag. Otherwise, the master sends an empty task to launch the finalization of the workers. Master finalizes (`MPI_Finalize`) when all results had been waited.

Listing 4.3 shows the source code of workers - `worker.c`. Lines 11 to 15 shows the MPI process sets up, where the intercommunicator with the master is get (`MPI_Comm_get_parent`), and the task and result data types are created. Then, each worker (lines 17 to 37) receives a task and verifies its content. When the received task is empty, the worker breaks the iteration and finalize (`MPI_Finalize`).

Listing 4.3: Pseudo code for a Master/Worker matrix multiplication: `worker.c`.

```
1  int main(int argc, char *argv[]) {
2    int myrank;
3    int i, j, k;
4    int K;
5    s_task_t input;
6    s_result_t res;
7    MPI_Comm m_comm;          /* intercommunicator with the master */
8    MPI_Status st;
9    MPI_Datatyepe task_t, result_t;
10
11   MPI_Init(&argc, &argv);
12   MPI_Comm_get_parent(&m_comm);        /* getting intercommunicator */
13   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
14   task_t = create_task_Datatype();
15   result_t = create_result_Datatype();
16
17   while(1){
18     /* receiving a task*/
19     MPI_Recv(&input, 1, task_t, MPI_ANY_SOURCE, MPI_ANY_TAG, m_comm, &st);
20
21     if(input != task_empty){  /* if there is work, compute */
22       res.sub_C = (double *)malloc(sizeof(double) * input.K * input.K);
23
24       for(i = 0; i < input.K; i++){
25         for(j = 0; j < input.K; j++){
26           for(k = 0; k < input.K, k++){
27             res.sub_C[i * input.K + j] += input.sub_A[i * input.K + k] *
28                                           input.sub_B[k * input.K + j];
29           }
30         }
31       }
32       res.I = input.I;
33       res.J = input.J;
34       res.K = input.K;
35       MPI_Send(&res, 1, result_t, myrank, TAG, m_comm);
36     }else break;              /* else, finalize the execution */
37   }
38
39   MPI_Finalize();
40 }
```

Otherwise, it allocates the results sub-matrix `res.sub_C` according to the input sub-matrices size (`input.K`), computes the multiplication (nested `for` loop), and returns the result (a variable of `s_result_t` data type).

Compared with SPMD programs, Master/Workers ones can easier support volatile processors taking advantage of the centralized master control of the workload distribution. Thus, while the master waits for workers results (Listing 4.2, lines 37 to 49), it also can check the availability of the processors and launch the adaptive actions. Here, can be used the same solutions described to SPMD programs: to call a procedure in each iteration (such as the `malleability_handler` in Figure 4.11) or to create a thread to check the processors availability.

When the number of processors increases, the master launches workers into the new processors and sends tasks to them (as shown in Figure 4.12). It is important to note that after the growth the master must update the number of workers to allow the new workers waiting (to add positions in `w_comm` array). Beside, in our example, the task counter also must be updated since the bag of tasks was implemented as an array. However, this last update can be avoided using a stack to implement the bag, thus, the next task is always the one that is on top. On the other hand, as all workers are spawned by the master, there is no communication restrictions since

the relationship between master and workers (including those created during the growth) is always a parent/children one.

When some processors must be released, the shrinkage action will finalize the workers running on required processors (remembering that the master must never be stopped as the restriction of malleable Master/Worker programs). There are two possibilities: *(i)* the master identifies the workers that must finalize and sends an empty task to them; or *(ii)* the workers perceive themselves (for instance, catching a signal) that they must stop the computation and finalize to release their processors.

In the first option, the master receives the requesting of processors (*e.g.*, a list of host names that must be released). It must be able to identify which workers are running on them. The advantage of this approach is that changes are only demanded in the master side, which must include the growth procedure. Whereas, workers source code keeps equal to non-malleable programs. However, the releasing of the processors can happen with a significant delay, once workers will only be able to process the empty task after finish the computation of the previous one.

The second option has no delay since workers perform the shrinkage action stopping immediately their computation and finalizing. However, the workers must send back to master the task that has not been computed (*e.g.*. with a flag sets on to identifying that it must be replaced in the bag). Once the master receives a non-computed task, it replaces it in the bag of tasks, registers that the sender worker is finalizing, and excluded it from the waiting list (`reqs` and `w_comm` arrays in Listing 4.2). Thus, this approach has a low delay when compared with the first one, but it requires changes on both, worker and master ones.

4.4.2 Example of MPI Application following the Explicit Task Parallelism

In Section 4.3, we introduced the main aspects to provide MPI programs able to deal with unpredictable changes in the application workload. In this section we illustrate these aspects aiming to provide an evolving MPI implementation or an explicit task MPI implementation of the matrix multiplication.

Listing 4.4 shows a D&C implementation of the matrix multiplication - `D_and_C.c`. We omitted the source code that get the input data and spawns the first `D_and_C` process as shown in Figure 4.13. Thus, the `D_and_C` starts getting the parent communicator to receive the input data (lines 34 to 37). Likewise in Section 4.4.1.2, the structures `s_input_t` and `s_output_t` encapsulates input and output data: matrices, their sizes and the initial coordinates.

The function `dec_mm` starts testing the size of the input matrices in relation to a predefined threshold. The threshold determines when a problem becomes trivial to be solved sequentially, and as consequence, it stops the recursive partitioning of the input data. When the size of the input matrices is less or equal to the threshold, the function `sequential` iteratively computes the multiplication. Otherwise, the input is partitioned - `divide_input`. This function divides the matrices A and B in such a way that four new inputs are created: matrix A is divided into two sub-matrices A_1 and A_2 with $\frac{n}{2} \times n$ elements each; B is also divided into two sub-matrices B_1 and B_2 but with $n \times \frac{n}{2}$ elements each; thus $A \times B = A_1 \times B_1; A_1 \times B_2; A_2 \times B_1$; and $A_2 \times B_2$. Figure 4.14 illustrates the partitioning of the inputs as well as the waited output.

Listing 4.4: Pseudo code for a D&C matrix multiplication: `D_and_C.c`.

```
1  void dec_mm(s_input_t *in, s_output_t *out){
2    int i, err[1];
3    s_input_t ch_in[4];
4    s_output_t ch_out[4];
5    MPI_Comm ch_comm[4];
6    MPI_Request reqs[4];
7    MPI_Datatype input_t = create_input_t(), output_t = create_output_t();
8
9    if(in->kjA <= THRESHOLD){ // testing if granularity is small enough
10     sequential(in, out);
11   } else{
12     divide_input(in, ch_in);
13     // spawning children
14     for (i = 0; i < 4; i++){
15       MPI_Comm_spawn("D_and_C", MPI_ARGV_NULL, 1, info, rank, MPI_COMM_SELF,
16                      &ch_comm[i], err);
17       MPI_Send(&ch_in[i], 1, input_t, 0, CH_TAG, ch_comm[i]);
18       MPI_Irecv(&ch_out[i], 1, output_t, MPI_ANY_SOURCE, MPI_ANY_TAG, ch_comm[i],
19                 &reqs[i]);
20     }
21     for(i = 0; i < 4; i++)
22       MPI_Waitany(4, reqs, &j, MPI_STATUS_IGNORE);
23
24     merge_matrices(out, ch_out);
25   }
26 }
27 /*=============================================================*/
28 int main(int argc, char **argv) {
29   s_input_t in;
30   s_output_t out;
31   MPI_Comm parent;
32   MPI_Datatype input_t = create_input_t(), output_t = create_output_t();
33
34   MPI_Init(&argc, &argv);
35   MPI_Comm_get_parent(&parent);
36
37   MPI_Recv(&in, 1, input_t, MPI_ANY_SOURCE, MPI_ANY_TAG, parent, &st);
38   dec_mm(&in, &out);
39   MPI_Send(&out, 1, output_t, 0, P_TAG, parent);
40
41   MPI_Finalize();
42 }
```

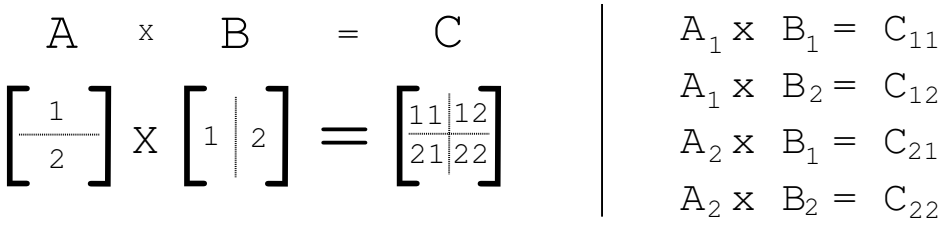


Figure 4.14: Division of input (A and B) and output (C) matrices.

After data partitioning, four new processes are spawned (Listing 4.4, lines 14 to 20): one to each operation showed in Figure 4.14 on the right. Then, as well as in Listing 4.2, the parent sends the input data and starts an asynchronous receives of the output. The parent blocks to wait for all children results through `MPI_Waitany` (lines 21 and 22). Then, `merge_matrices` function merges the sub-matrices with the partial results (C_{11}; C_{12}; C_{21}; and C_{22}) to compose the output.

The number of processes that will be created by this D&C matrix multiplication depends on two parameters: the threshold value and the size of the input matrices. Thus, the application can adapt itself according to its input size: the larger input size, greater the number of processes spawned. In other words, the application unfolds the parallelism at runtime, spawning processes to compute part of the input data concurrently. Furthermore, the same algorithm structure can be used to extract the parallelism with any input size.

However, the explicit task implementation of an MPI matrix multiplication showed in Listing 4.4 rises on some open issues. For instance, the granularity of the processes, or the ratio between computation and communication, depends of the threshold chosen, which defines when the computation can be performed sequentially. Furthermore, the granularity has different weights as the application phases: dividing or the partitioning of the data; computing or the iteratives multiplication; and conquering or merging of the data.

Another issues is related to mapping the dynamic processes *i.e.* in which available processors the new processes will be placed. Moreover, each level of recursion creates four processes, thus the number of processes increases exponentially 4^l being l the number of recursion levels. Furthermore, for some MPI distributions such as OpenMPI, when a processor becomes oversubscribed (with more processes than processors or cores) the performance of the programs goes down. Thus, all these issues show that the development of MPI applications with a paradigm close to the explicit task parallelism requires some cares to ensure a good performance. Chapter 6 will present our considerations about the performance of this kind of evolving MPI programs.

4.5 Conclusion

This chapter intended to answer the question: "*How to provide adaptability using MPI?*". Thus, it started describing the dynamic process creation, its features and impact in programs development. Then, Section 4.2 described how to dynamic MPI processes can be used to allow an on-the-fly adaptation when the processors have a volatile availability. To increase the number of used processors, malleable MPI applications spawn new processes into further processors; and to decrease the number of processors, processes into the required processors are finalized. In addition, it was described the required interactions between MPI applications and RMS systems to exchange information about the processors availability, and the development issues to provide malleable MPI applications as SPMD and Master/Worker program structures.

Aiming to deal with unpredictable needs, Section 4.3 showed how to take advantage of the dynamic process creation to unfold the parallelism at runtime as the target architecture. The section was focused on D&C algorithms and it was defined the abstract MPI tasks, how to solve their dependencies through message passing synchronizations, and how to achieve load balancing through the granularity control. To clarify issues about the development of adaptive MPI programs, Section 4.4 presented the matrix multiplication problem as well as its malleable implementation as SPMD and Master/Worker program structures and its explicit task implementation as D&C algorithm structure (Fork/Join program structure).

The conclusion of this chapter is that the dynamic process creation can be used in MPI programs to provide adaptability to volatile processors (malleable applications), as well to unpredictable needs (explicit task applications). However, each class of application has their own requirements, including runtime environments ones, which must be provided to ensure the applications performance and correctness. The next two chapters show the treatment of these requirements in practice. Thus, Chapter 5 focus on supporting issues of the malleable MPI applications, while Chapter 6 focus on explicit task ones.

5 RUNNING MALLEABLE MPI APPLICATIONS IN CLUSTERS

In the previous chapter, we show the requirements to allow the development and execution of malleable MPI applications (Section 4.2). Furthermore, we show how to use the MPI-2 features to implement these applications through a matrix multiplication examples (Section 4.4.1). We could see that the main issues to support malleability in current parallel environments are:

- An RMS system able to deal with malleable applications, *i.e.* it must decide the processors allocation taking into account the requirements of running malleable applications;
- To ensure that adaptive actions provide expected reactions as happen changes in processors availability, *i.e.* to ensure the correctness of the growth and shrinkage actions;
- To provide information exchange between malleable MPI applications and RMS systems aiming to keep both updated.

This chapter aims to show the highlighted issues in practice, allowing the execution of the malleable MPI applications in cluster architecture. In this sense, Section 5.1 describes the RMS systems and their support to the requirements of malleability as well as the RMS used in our experiments: OAR. In the following, Section 5.2 describes features provided by the MPI distributions as well as their use to support malleability. Furthermore, it shows a scheduling library proposed by us aiming to provide the mapping of dynamic MPI process and its use in malleable MPI applications. This scheduling library and the RMS system must interact to exchange information about the volatile processors, thus, the means used to implement these interactions will be also exposed. Finally, Section 5.3 shows the experimental results and analyses the performance of a malleable MPI application on clusters.

5.1 RMS and the Management of Volatile Processors

Section 3.1 exposed some initiatives that use specific middlewares to manage volatile processors and support the execution of malleable applications. Excluding these initiatives and according to our up-to-date knowledge, there is no existing implementation of malleable jobs support upon a generic resource manager since it is a rather complicated task.

Nevertheless, some works have studied specific prototypes aiming at malleability with different constraints (HUNGERSHFÖER; STREIT; WIERUM, 2001; UTRERA; CORBALÁN; LABARTA, 2004). For instance, UTRERA; CORBALÁN; LABARTA (2004) propose the Folding by JobType (FJT) technique that can vary the number of processors used by a parallel job without requiring changes in the application source codes. FJT associates three concepts:

- **Moldability** to decide the number of processors at starting time;
- **Folding** to provide malleability;
- **Co-Scheduling** to allow the sharing of the processors by some processes.

Thus, folding allows varying the number of processors at runtime: applications can fold to release half of used processors; otherwise, they expand to duplicate the current number of processors. In this way, folding happens in the runtime system level being transparent to applications. Although this research dates from 2004 and, as far as our knowledge, it was not continued by the authors, the use of folding to provide malleability is attractive and can be implemented using recent features of the Operating Systems (OS) such as the handle of the `CPUSET`s. Our research partners studied this and additional information can be found in CERA et al. (2010).

In HUNGERSHFÖER; STREIT; WIERUM (2001), a middleware called Application Parallelism Manager (APM) provides on-line scheduling of malleable jobs on shared-memory architectures (SMP). This approach combines advantages of time and space sharing scheduling. They achieved 100% system utilization and a small response time in their simulations. Nevertheless the system was not directly adaptable to distributed-memory machines with message-passing and this work, as far as we know, keeps on simulation context.

Under this context, we investigate the support of malleable jobs considering a generic resource management system: OAR. Figure 5.1 illustrates the support awaited from OAR resource manager: to execute malleable applications in a cluster environment, the RMS must provide the management of volatile processors as well as ways to implement flexible job allocations allowing changes in their number of processors during runtime. These requirement will be provided by OAR as will be described in the next section.

5.1.1 The OAR Resource Manager

OAR[1] (CAPIT et al., 2005) is an open source resource manager for large scale clusters that has been developed at *Laboratoire d'Informatique de Grenoble* (LIG). It provides the main issues awaited for resource managers such as reservation, prioritize jobs, resources matching, backfilling, etc. Thus, OAR is a robust solution, used as a production system in various cases, for instance, in the management of Grid5000 (BOLZE et al., 2006), and Ciment[2] infrastructures. Due to its open architectural choices, based on high level components (Database and Perl/Ruby Programming Languages), OAR can be easily extended to integrate new features and treat research issues as those desired in our work.

Moreover, OAR offers *Best Effort* jobs that are a special type of job aiming at improvements in processors utilization (GEORGIOU; RICHARD; CAPIT, 2007). They are defined as jobs with minimum priority, able to harness the idle processors

[1] *http://oar.imag.fr/* - last access in January 2011
[2] *https://ciment.ujf-grenoble.fr/cigri* - last access in January 2011

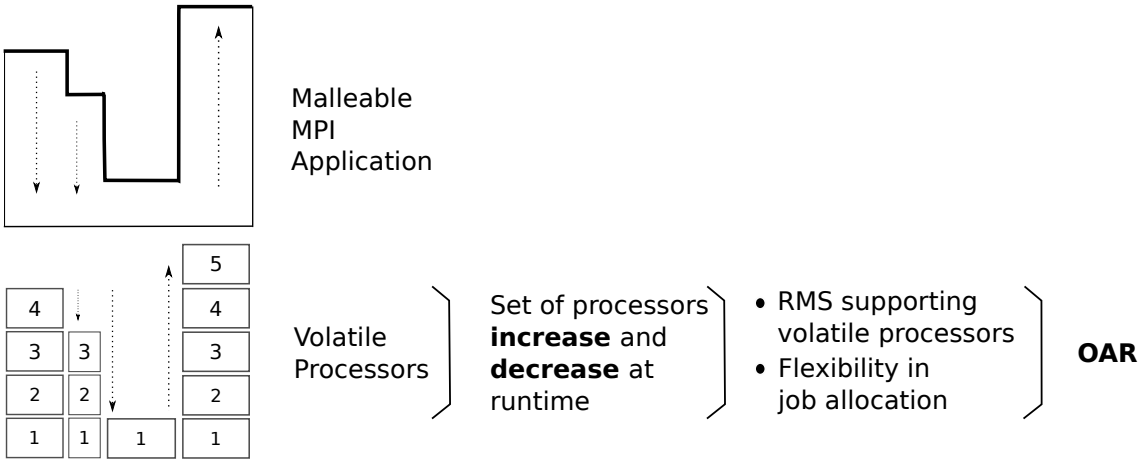


Figure 5.1: Requirements of malleability in RMS level: *(i)* management of the volatile processors; and *(ii)* ways to include some flexibility in job allocation allowing increasing and decreasing the number of processors at runtime.

of a cluster, but they will be directly killed when processors are required. The idea of *Best Effort* jobs is comparable with the notion of 'cycle stealing' initiated by Condor (LITZKOW; LIVNY; MUTKA, 1988) and the High Throughput Computing approach. *Best Effort* jobs rise important features that can be used to support malleability: the allocation of processors can be more flexible in such a way that some can be added or released at runtime. In our research, we take advantage of this kind of job to support the execution of malleable MPI applications.

Besides *Best Effort* jobs, the OAR team develop research to provide more flexible processors allocations, *i.e.* its scheduling decisions are able to change the allocation of the processors during the applications execution improving the resource utilization as well as the response time. These issues were investigated in a PhD thesis (GEORGIOU, 2010) at LIG. We cooperate with the French group in such a way that our malleable applications use the volatile processors provided by OAR. This cooperation helped to identify and solve issues in both fronts. Furthermore, a CAPES/Cofecub International Cooperation Project between both research groups supported this cooperation. This project supported my stage (sandwich scholarship) in LIG laboratory during one year, and this chapter describes the main results achieved in the cooperation.

5.1.2 Management of the Volatile Processors in OAR

In OAR, the management of volatile processors intents to improve the overall utilization of the clusters. In this sense, OAR takes advantage of the malleable jobs, running them along with rigid ones (*i.e.* the standard jobs managed by OAR) on their unused processors. This is a known practice to improve cluster utilization (HUNGERSHÖFER, 2004; GHAFOOR, 2007; BUISSON et al., 2007). Due to the modularity and flexible implementation of OAR, its team was able to construct a prototype, aside of OAR core, which implements the support of malleable jobs. Thus, the complexity and advantages of this strategy could be investigated before the development of a production solution. We use this prototype in our experiments.

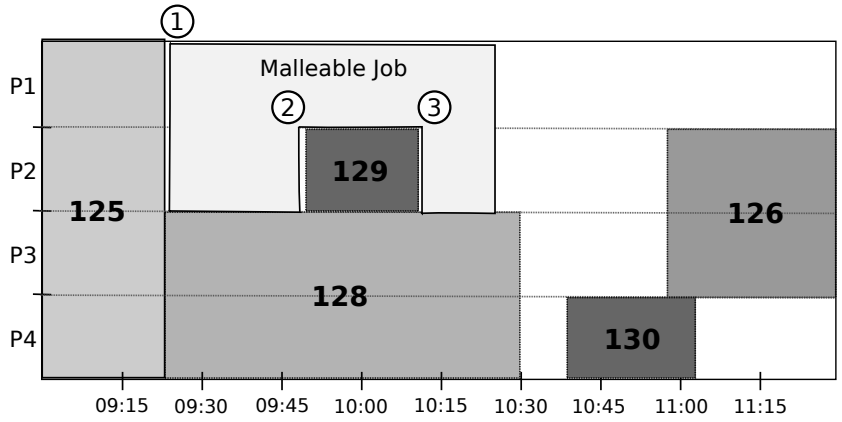


Figure 5.2: The behavior of a malleable job executing together with rigid ones. Three scheduling decisions are required: *(1)* to identify the unused processors at malleable job starting time; *(2)* to request one processor to serve a new job; and *(3)* to provide a further processor that was released by a concluding job.

The prototype updates the OAR scheduling policy in such a way that if there are malleable jobs available, they will execute on free processors of clusters. Free processors are those that are not being used by rigid jobs and will keep idle until an eventual allocation. Malleable jobs run with low priority while rigid ones have high priority in OAR. The number of processors of malleable jobs, as well as the variation on this number, is driven by the rigid jobs requirements:

- When a malleable job is submitted, OAR regards the current and near future jobs allocations assigning all the unused processors to it. This provides the initial amount of processors, which must be between the minimum and maximum number of awaited processors by the malleable job;
- The number of processors of a malleable job change when:
  - **A new job starts.** In this case OAR requests some processors from malleable jobs to attend to new job requirements. OAR decides how many processors each malleable jobs must release taking into account their minimum required number;
  - **A running job concludes.** The processors released by the concluded job may be assigned to malleable jobs which have less than their maximum number of processors.

Notice that changes in the set of processors of malleable jobs happen when any kind of job (malleable or rigid) concludes or starts. Furthermore, OAR takes decisions considering the malleable jobs requirements (minimum and maximum number of processors) as well as the rigid ones (required processors). The version used in our experiences supports only one malleable job at a time to avoid issues on scheduling processors among several malleable jobs. The OAR team intends to support multiple malleable jobs in future works.

To identify the free processors of an infrastructure, the OAR team implemented a resource discovery command. This command provides current and near future available processors and it is called at malleable jobs starting time as well as when adaptive actions are required. Basically, the resource discovery command queries the OAR database identifying the current unused processors. Furthermore it looks

at the reservations to avoid the assignment of processors that will be required soon by future jobs. The default time that OAR looks forward in reservations is 15 minutes, but this time slice can be changed according to features of the malleable jobs. Although this strategy reduces the chances of release processors soon after the malleable job starts to use them, it does not avoid the releasing when iterative jobs are submitted. Iterative jobs are another kind of job offered by the OAR, which have high priority and they start as soon as they are submitted, since there are enough processors to attend them.

Figure 5.2 illustrates the execution of a malleable job in OAR. Supposing that a malleable job was submitted during the execution of the *job* 125, it will only start when some processors become unused. Thus, *(1)* represents when OAR identifies 2 free processors (through the resource discovery command) and allocates them to the malleable job. In *(2)*, OAR decides that the malleable job must release one processor that will be allocated to the new job (*job*129). Finally, in *(3)*, OAR allocates another processor to the malleable job, since one of the running jobs concludes (in our illustration is the same *job* 129). Then the malleable job continues to execute until its completion. Notice that if another malleable job is waiting to start immediately after the first one, the resource discovery command will identify 3 unused processors ($P1$, $P2$, and $P3$), since the processor $P4$ was reserved to *job* 130, which will start before the 15 minutes that OAR looks forward.

5.1.3 Providing Malleable Jobs in OAR

Malleable jobs in OAR are composed by two parts: rigid and *Best Effort* ones. The rigid part represents the minimum requirements of a malleable application, *i.e.* its invariable part that must never be released. Thus, this part is a rigid job (a standard OAR job) and stays running during all application lifetime, so that it can guarantee the job completeness. In Chapters 2 and 3 we introduced that malleable jobs requires a minimum and maximum number of processors to execute. The rigid part represents the minimum requirement. For instance, to a Master/Worker application, the minimum number of processors is 1, in which at least the master must always be running. In an OAR malleable job, the rigid part will have the master running on a processor that will never be released during the application execution.

The *Best Effort* part allows a dynamic behavior in OAR malleable jobs taking advantage of the *Best Effort* job features. It includes everything that is not in the rigid part of the malleable job, furthermore the size of the *Best Effort* part is defined at starting time and can vary during the application execution. To allow a malleable behavior at runtime, each processor of the *Best Effort* part runs one *Best Effort* job. For instance, if the *Best Effort* part has 5 processors, there are 5 *Best Effort* jobs running. Thus, this part of the malleable job can vary as the availability of the processors, *i.e.* as the results of the resource discovery command:

- **When the number of processors was increased:** further *Best Effort* jobs are submitted, one to each added processor, increasing the size of the *Best Effort* part of the job, *i.e.* the malleable job grows;
- **When the number of processors must decrease:** the *Best Effort* jobs running on the required processors are killed, releasing the processors and decreasing the size of the *Best Effort* part, *i.e.* the malleable job shrinks.

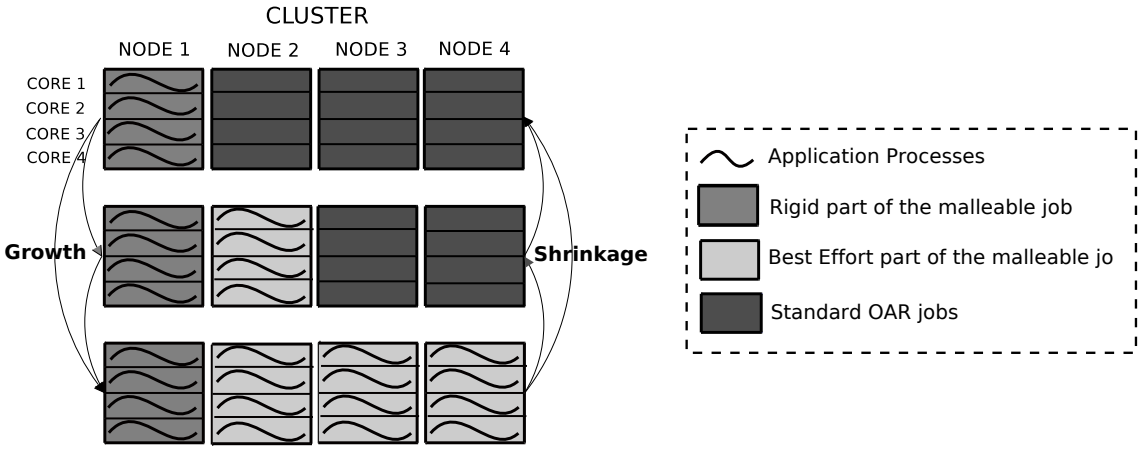


Figure 5.3: Illustration of an OAR malleable job in a cluster with 4 quad-core processors: in the left, arrows are representing the growth; and in the right, arrows representing the shrinkage.

OAR provides malleable jobs through bash scripts, thus they are transparent to the users and do not request special care about their two parts. Furthermore, note that when some processors must be released causing the killing of some *Best Effort* jobs, this requires a reaction in application to avoid crashes. In our experiments, processes running on required processors will be finalized and this can take some time. Thus, OAR adds a grace time delay before kill the *Best Effort* job to allow the finalization of the processes without failures. A grace time delay represents the time slice waited by OAR before destinate a processor to another job, ensuring that they are free.

Figure 5.3 illustrates an OAR malleable job running in a cluster with 4 quad-core processors. *node* 1 has the rigid part of the malleable job, which computes 4 processes of the application to take advantage of the multi-core infrastructure. The arrows in the left represent the growth action, in which further *Best Effort* jobs are submitted allowing to increase the current number of processors at runtime. In the right, the arrows represent the shrinkage action where *Best Effort* jobs running on the required processors are killed to release them.

This section described issues to support malleable jobs on OAR resource manager. In the following section we will describe the application requirements to allow malleability.

5.2 MPI Application dealing with Volatile Processors

In Section 4.2, we showed that malleable MPI applications require:

- An interface between the malleable application and the RMS system allowing notifications about the processors availability;
- To implement a procedure to identify changes in processors availability as well as the procedures to grow (through dynamic process creation) and shrink (through the finalization of processes) the application;

However, these requirements bring other issues such as the support of the MPI distributions either to create dynamic processes or to allow to change the number of processors at runtime. Moreover, to decide the process mapping on-the-fly is a key.

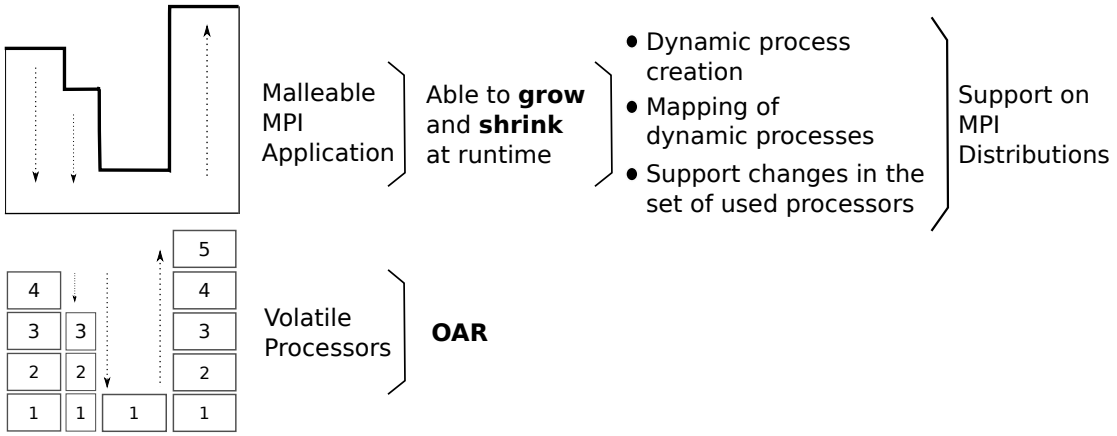


Figure 5.4: Requirements of malleability in application level: *(i)* to support dynamic process creation; *(ii)*to map dynamic MPI processes; and *(iii)* means to change the number of used processors at runtime.

Figure 5.4 illustrates the requirement expected from MPI distributions to provide malleability in MPI applications. Malleable MPI applications are able to grow and shrink at runtime as a reaction to changes in processors availability. To implement this behavior, it is expected that the MPI distributions implement the dynamic process creation (`MPI_Comm_spawn` and correlated primitives) as well as allow to map them at runtime. Furthermore, the number of processors used by MPI applications must be able to vary at runtime, *i.e.* the application must be able to identify changes in its set of processors and do not crash after changes.

Thus, this section is divided into: Section 5.2.1 that describes how MPI distributions deal with changes in the set of used processors; Section 5.2.2 presenting the default mapping offered by the MPI distributions and their restrictions; Section 5.2.3 that introduces our scheduling library to map dynamic processes; Section 5.2.4 shows the changes on the scheduling library aiming to deal with volatile processors; and Section 5.2.5 shows the implementation of the interactions between OAR and our scheduling library to allow the execution of malleable MPI applications in cluster environments.

5.2.1 Malleability support on MPI Distributions

Nowadays, dynamic process creation – `MPI_Comm_spawn` and correlated functions (saw in Section 4.1.2) – is implemented by most current MPI distributions. However, in some cases, the distributions fall short in issues about the support of volatile processors. For instance, when a new processor becomes available, it must be known or accessible by the others (*i.e.* those running before the addition). Thus, MPI distributions must offer some way to update the applications about changes in the set of used processors.

This section aims to show how these issues are provided in the MPI distributions. We selected two study cases: LAM/MPI (Section 5.2.1.1); and OpenMPI (Section 5.2.1.2). As our up-to-date knowledge, these distributions exemplify the scenarios commonly found in the current MPI distributions.

5.2.1.1 ***LAM/MPI***

Using LAM/MPI, there are two ways to launch an MPI application:

- Running the `lamboot` command, providing as argument a list of host names, and then launching the application (through `mpirun`/`mpiexec` commands);
- Using the options `-machinefile <file>` or `-hostfile <file>` of `mpirun` / `mpiexec`, in which the `<file>` has the list of host names. This option provides the `lamboot` transparently to the users.

In both cases, a `lamd` daemon is started in each node (processing elements are called nodes in LAM/MPI, which can have more than one CPUs or cores) to compose a network of daemons allowing to launch MPI processes on them. Thus, an MPI application only knows the nodes that are part of its network.

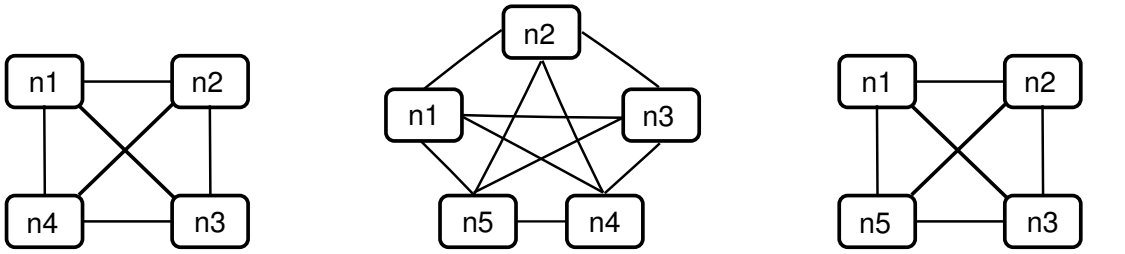
LAM/MPI provides two commands that allow changing the current number of nodes of a LAM/MPI network:

- `lamgrow [-cpu <num>] [-n <nodeid>] <hostname>` (briefly introduced in Section 4.1.4);
- `lamshrink [-w <delay>] <nodeid>`.

The first command provides the growth of the LAM/MPI network: it launches a `lamd` daemon in the new node connecting it with the others. The host name is given by the `<hostname>` argument, as well as the optionals: number of CPUs (`-cpu <num>`); and the node identifier `-n <nodeid>`. However, notice that the `lamgrow` only acts on the LAM/MPI network without providing any notification to the running MPI processes. One specific LAM/MPI function (`getntype()`) allows to query the LAM/MPI network to know about its current number of nodes tanks to special mask (`0x02`). Thus, calling this function, MPI applications can verify the current number of processors and take the appropriate reaction.

The second command allows the shrinkage of the LAM/MPI network: it kills the `lamd` daemon running in the required node without crash the LAM/MPI network. The `lamshrink` command requires the identifier of the releasing node as argument. Furthermore, `lamshrink` provides a mechanism to delay the killing of the `lamd` daemons. This delay is useful to give some time to MPI applications to react to the future loss of nodes. When calling `lamshrink` with `-w <delay>` option, the command warns the MPI processes on the target node about the impending release through a signal (`SIGFUSE`). Then the command pauses for the given `<delay>` seconds before removing the node from the LAM/MPI network. Thus, the MPI processes that catch the signal may finalize their execution while they are still connected to the remaining application.

Figure 5.5 illustrates the LAM/MPI network after the execution of the commands previously described. In the left, when the user calls `lamboot myhosts`, having four host names in `myhosts` file, a `lamd` daemon is launched in each host composing the illustrated network. After some time executing with four nodes, the user calls `lamgrow -n 5 newhost` to add a new host in the network. When it is finishes the command, the `n5` node was included in the network being accessible by the MPI application (in the middle). Then, the user requires the exclusion of `n4` through `lamshrink -w 10 n4`. This command warns the processes on `n4` and will wait for 10 seconds before releasing the node. When `lamshrink` returns, the node `n4` is no longer part of the LAM/MPI network.



`$lamboot myhosts` `$lamgrow -n 5 newhost` `$lamshrink -w 10 n4`

Figure 5.5: Managing the LAM/MPI network of `lamd` daemons: *(i)* `lamboot` to startup the network using the host set on `myhost` file; *(ii)* `lamgrow` to increase the number of nodes at runtime – adding the `n5` node; and *(iii)* `lamshrink` to release nodes at runtime – removing the `n4` of the network.

5.2.1.2 ***OpenMPI***

OpenMPI also has a daemon running in each node used by MPI applications, composing a network of daemons. This daemon is called `orted` and runs as a message router. However, OpenMPI does not provide a special command to boot the network (like `lamboot` in LAM/MPI), being started through `mpirun`/`mpiexec` commands with `-machinefile <file>` or `-hostfile <file>` options. Thus, the startup of the network of daemons happens transparently for the users (along with the application launching).

OpenMPI does not provide any special command to increase or decrease the number of nodes at runtime (like `lamgrow` and `lamshrink` previously introduced). However, changes in the network are viable since some constraints are taken into account. When an MPI process is spawned into a node out of the network of daemons, this causes the background launching of the `orted` daemon before the spawning of the process. Thus, the OpenMPI network increases at runtime. But to allow this on-the-fly increase, the host name of all candidates to be add at runtime must be provided at starting time in the `<file>` argument of the `mpirun` or `mpiexec`. In other words, the application requires the host names of all its potential nodes at starting time, including those that will only be used at runtime.

OpenMPI does not provide signal and wait mechanisms like LAM/MPI, but this can be externally provided. Basically, when a node must be removed from the OpenMPI network, we provided a procedure that warns the MPI processes on the node through a signal (for instance, `SIGTERM`). Then a certain time is waited to ensure the processes finalization. However, this approach raises a collateral effect: once a `orted` daemon was started, it cannot be stopped before the application finalization. This is because if an `orted` deamon is stopped, it breaks the message routing causing a crash in the MPI application. According to OpenMPI developers team, some research on fault tolerance is being done aiming to allow the stopping of some `orted` daemons without crash the entire network. As there are no perspectives of when such feature will be available, the exclusion of nodes can be provided keeping the `orted` running on released nodes. Furthermore, `orted` consumes little CPU power, thus it does not compromise the node performance when it is allocated to other users.

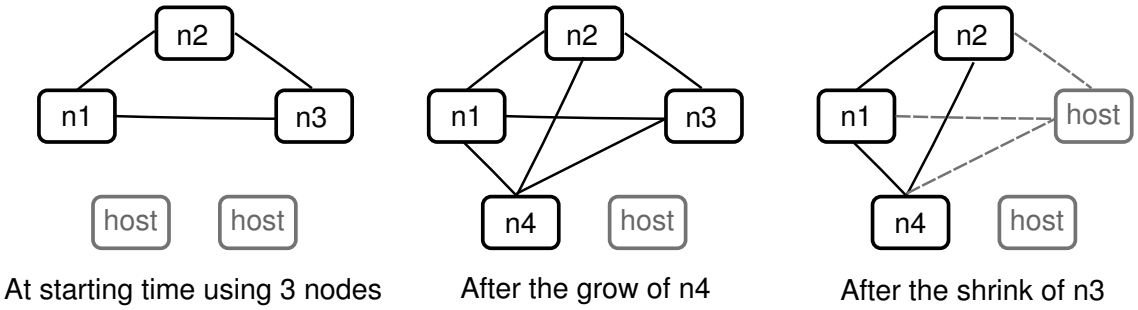


Figure 5.6: Managing the OpenMPI network of `orted` daemons:*in the left,* the application starts knowing five nodes, but uses only three that run `orted` daemons; *in the middle,* a fourth node starts to be used through the spawning of processes on it (growing of the network); *in the right,* the `n3` node is requested and after its MPI processes finalize, it keeps running the `orted` daemon to avoid loss of messages.

Figure 5.6 illustrates the OpenMPI network of `orted` daemons. At application starting time, the user provided a file with five host names, but the application only uses three of them, thus only these three hosts are part of the OpenMPI network, having `orted` daemons running. Then, a further node is added in the network (`n4`) through the spawning of dynamic processes on it. Before using the further node, OpenMPI launches an `orted` and the node becomes known to the others. Finally, when the `n3` node is requested, the processes running on it are warned and after some time the node has no MPI processes, but `orted` keeps running to allow the routing of messages.

5.2.2 Issues on Mapping Dynamic MPI Processes

Previously, we described the two most common scenarios of volatile processors support on MPI distributions. Once they allow changing the current number of processors, this feature impacts on MPI process mapping, which is an important issue even in static environments. To guide the dynamic process mapping, MPI defines a data structure `MPI_Info` allowing informing, at runtime, how and where new processes must be spawned (fourth argument of `MPI_Comm_spawn`– saw in Section 4.1.2). The information is provided through key-value pairs, in which the MPI standard defines a set of frequent used keys (for instance, `host` - the host name to launch the processes on, and `machinefile` - file with a list of host names) and the users can define their own keys.

In the following, we will describe how the MPI distributions deal with dynamic process mapping issues and how the users can take advantage of `MPI_Info` structure.

5.2.2.1 ***LAM/MPI Mapping***

LAM/MPI offers a Round Robin strategy to map the dynamic MPI processes: given a circular list of hosts, when a new process must be placed, it always goes to next host after the last that received a process. Thus, when all hosts received one process, the strategy follows the same sequence to assign the second and so on. Figure 5.7 illustrates it: there is a circular list of host and a pointer to the next host able to receive a process. When a process is assigned, the pointer is updated. LAM/MPI allows that users set who is the first host in a Round Robin distribution through a specific key of `MPI_Info` defined by LAM/MPI: `lam_spawn_sched_round_robin`.

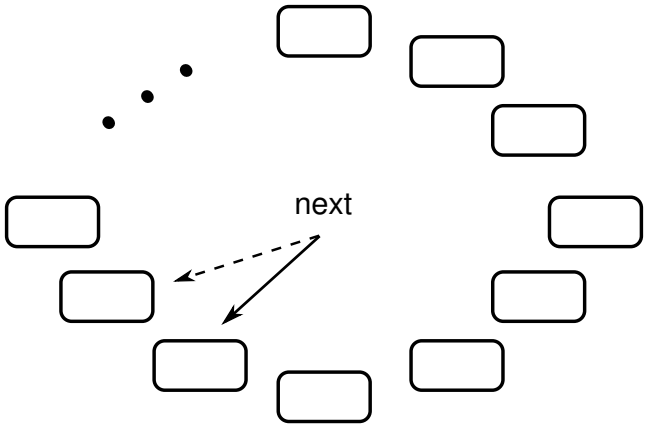


Figure 5.7: Round Robin process mapping: a circular list of the host is followed assigning one process to each host; `next` pointer indicates the next host that must receive a process and it is updated in each assignment.

Aiming to test the dynamic process mapping of LAM/MPI, we developed a simple MPI program that spawns 20 processes implemented in two ways:

1. A loop calls `MPI_Comm_spawn` 20 times, creating one process at each time;
2. An unique `MPI_Comm_spawn` call to create 20 processes.

Thus, the difference is the third argument of `MPI_Comm_spawn`, which is set as 1 in the first test, and as 20 in the second one. Table 5.1 shows the achieved distribution. As can be seen, LAM/MPI Round Robin only works in the second test, *i.e.* when all processes are launched from an unique `MPI_Comm_spawn` call.

Table 5.1: Testing the LAM/MPI Round Robin mapping of dynamic processes.

	Node 1	Node 2	Node 3	Node 4	Node 5
20 spawns of 1 process	20	0	0	0	0
1 spawn of 20 processes	4	4	4	4	4

In the first test, LAM/MPI places all processes in the same node, which is the first one in the host list. This means that LAM/MPI is not able to keep the pointer information from one execution to another. This undesirable behavior can be solved storing this information in the application source code. To each `MPI_Comm_spawn` call, the `lam_spawn_sched_round_robin` key is set with the next host in the list. Thus, both approaches can provide a balanced distribution of the dynamic MPI processes. We described this LAM/MPI issue, as well as proposed a library to solve it transparently to the users in CERA et al. (2006) and CERA et al. (2006). Section 5.2.3 will describe this library.

Notice that all the previous considerations about LAM/MPI only take into account the dynamic process mapping on static environments, *i.e.* without changes in processor availability. However, once the LAM/MPI allows to change the number of processors at runtime through `lamgrow` and `lamshrink` commands, the same issues are valid in a dynamic context. As the components of LAM/MPI network can vary at runtime, the dynamic process mapping will take changes into account. Furthermore, in Section 5.2.4 we will describe how these commands are used in the management of the volatile processors.

5.2.2.2 ***OpenMPI Mapping***

In OpenMPI the default mapping of dynamic MPI processes follows the Round Robin strategy. Furthermore, it is able to take into account the number of processors or cores of the nodes during the process distribution, in such a way that each processor or core receives one process. To OpenMPI, the same simple test performed to LAM/MPI achieved a balanced distribution of the dynamic processes for both tests: to multiple calls of `MPI_Comm_spawn` as well as to one `MPI_Comm_spawn` creating multiple processes.

However, OpenMPI has two different executing modes that impact in the application performance: aggressive and degraded. The aggressive mode is achieved when the number of running processes is equal or less than the number of cores/processors in a node. Thus, the node executes *exactly-* or *under-subscribed* in such a way that each process will never give up voluntarily the processor during its execution. In other words, this provides a kind of processor-affinity between MPI processes and processors/cores, and thus, OpenMPI can ensure better performance.

On the other hand, OpenMPI falls in degraded mode when there are more processes than available processors/cores. In this case, the nodes are *oversubscribed*, *i.e.* processes will voluntarily give up the processor causing some loss of performance (context switch and message routing). Figure 5.8 illustrates processors as rectangles that are composed by two cores (rectangles partitioned in two parts) and the three states as the process mapping.

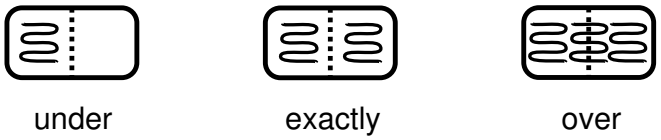


Figure 5.8: States of the nodes to OpenMPI: *under-subscribed* - the number of running processes is less than the number of cores; *exactly-subscribed* - there are as many processes as cores; and, *oversubscribed* - there are more processes than cores.

OpenMPI is able to provide the expected mapping of dynamic processes in a static environment, but it has no support to manage volatile processors during the applications execution. Although this management can be externally provided (as saw in Section 5.2.1.2), it requires the notification of the MPI processes about changes to allow the coherent reaction in the application side. Thus, OpenMPI can provide the mapping of dynamic processes, but it requires some external support to volatile processors. Section 5.2.4 exposes our proposed solution to provide malleability using OpenMPI.

5.2.3 A Scheduler for Dynamic MPI Processes

When we started to research the development of MPI applications able to spawn processes at runtime, only the LAM/MPI distribution was able to support this feature. Thus, we opted to use LAM/MPI, but in our first experiments, we have found an unexpected behavior in dynamic process mapping described in Section 5.2.2.1. Thus, our first effort was focused on improving the process distribution in distributed-memory environments such as clusters of computers. We implemented a scheduler to map the dynamic MPI processes, which will be described in this section.

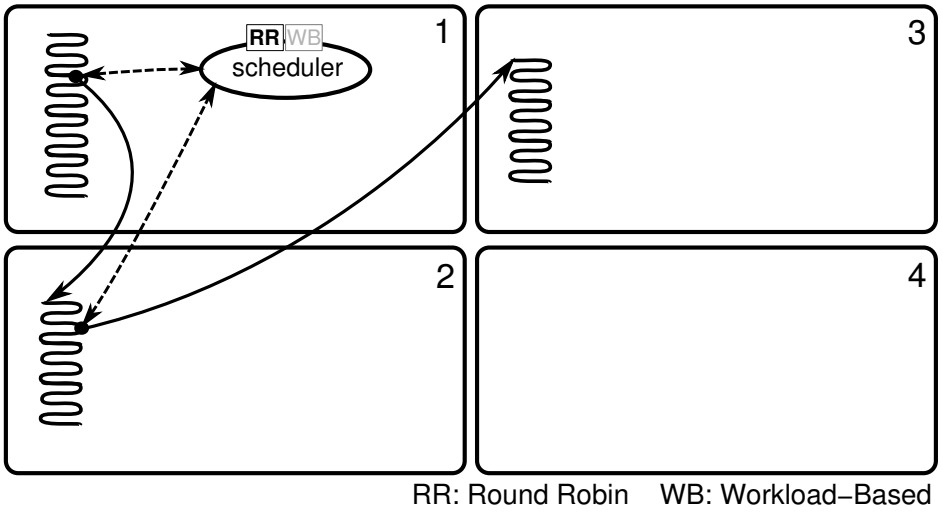


Figure 5.9: Scheduling library for dynamic MPI processes: when there is an `MPI_Comm_spawn` in the application source code, the scheduler daemon receives connections from the MPI processes that must know where spawn new processes (dotted arrows). Then, new processes are placed according to the scheduler decision – the solid arrows represent the dynamic process creation. It is shown the Round Robin strategy of process mapping (*node* 1 to *node* 2, and *node* 2 to *node* 3).

The scheduler is implemented as a user-library able to identify dynamic process creation and provide which will be the physical location of new processes (CERA et al., 2006). Our first objective was to solve the LAM/MPI problem, thus the scheduler implements a Round Robin distribution of the dynamic processes, as will be described in Section 5.2.3.1. Furthermore, we also implemented a strategy based on processors workload, aiming at MPI-2 applications with irregular load (see Section 5.2.3.2). The user can select between one of these mapping strategy at compile time by setting the proper flags variables.

Basically, our scheduling library is composed by an interface that provides functions to be used by the programmers and a scheduler daemon. To use the library, the programmer must include its head in the MPI application source code (`# include <lib-dynamicMPI.h>`). Then, at compile time, every call to `MPI_Comm_spawn` will be overloaded by its library version. Library's `MPI_Comm_spawn` version includes a procedure that queries the scheduler daemon asking where a new process must be placed. Once the answer is available, the `lam_spawn_sched_round_robin` key of `MPI_Info` is set, and the dynamic process creation is performed. As this key informs where LAM/MPI must start the Round Robin distribution, the new process will be placed as expected. The overloading of the `MPI_Comm_spawn` allows transparent interactions between MPI application and scheduler daemon, *i.e.* without change the structure of MPI application source code.

The scheduler daemon was implemented as an MPI process that runs along with the MPI application. This decision makes the information exchanging between scheduler and processes that are performing `MPI_Comm_spawn` easier. Thus, when an MPI process must know where to spawn its child, it can establish a client/server communication with the scheduler daemon (this kind of communication was explained in Section 4.1.3) and perform `MPI_Send`/`MPI_Recv`. As the mapping strategy set up, the daemon decides the physical location of new processes always that it is queried.

Figure 5.9 illustrates how our scheduling library works during the MPI application execution. When an MPI process reaches an `MPI_Comm_spawn`, it connects to the scheduler daemon requesting the physical location of the new process (dotted arrows). When the answer is available, the processes are spawned (solid arrows). In this illustration, the Round Robin policy was adopted, since the new processes are mapped from *node* 1 to *node* 2, and *node* 2 to *node* 3.

5.2.3.1 ***Round-Robin Mapping***

The scheduling library implements the Round Robin strategy keeping a reference to the last used host in the host list, which follows strictly the same sequence of the host names provided at application starting time. Thus, when the scheduler daemon is requested during a process spawning, it computes:

$next\_resource = ((last\_resource + 1)\%total\_resources)$

and sends back the *next_resource*. Then, the *last_resource* stores the value of *next_resource*. As the *last_resource* information is stored from one request to another, the daemon is always able to determine who the next host as the Round Robin strategy is. The content of this section was published in CERA et al. (2006).

To test the process mapping of our scheduling library Round Robin strategy, we implemented the Fibonacci computation (briefly introduced in Section 3.2.1). The program returns the ith element in the Fibonacci sequence $(0, 1, 1, 2, 3, 5, 8, 13,...)$, being each element the sum of the previous two. A recursive implementation calls the Fibonacci function to search the $n - 1$ and $n - 2$ elements until n becomes less than 2, since the Fibonacci of $n < 2$ is equal to n. In the MPI-2 implementation of Fibonacci, the recursive calls are replaced by dynamic process creations, thus a new MPI process is spawned to compute each value of n. Although this is not the most efficient implementation of Fibonacci, since many recalculation are performed, it is widely used as a didactic example and allows to test the dynamic process mapping.

Table 5.2 shows the distribution of the Fibonacci processes upon 5 nodes of a cluster aiming to calculate the $6th$ element in the sequence. As can be observed, the standard LAM/MPI distribution is not able to distribute the processes among the available nodes. However, the same source code using the scheduling library achieves the expected process mapping.

Table 5.2: Number of spawned processes per node to calculate $6th$ element of Fibonacci, using the Round Robin strategy of LAM/MPI and our scheduling library.

	Node 1	Node 2	Node 3	Node 4	Node 5
LAM/MPI	25	0	0	0	0
Scheduling library	5	5	5	5	5

Although the Fibonacci calculation is a good test to evaluate the distribution of the processes since it requires many process creations at runtime, it is not a good performance metric. This is because each Fibonacci process performs few computations: it spawns two processes (to compute $n - 1$ and $n - 2$, respectively); waits for their results; and sum the returned values. Thus, we tested the scheduling library with an application that search for prime numbers. It computes on an interval of number from 1 until N and is implemented as a Divide and Conquer algorithm structure. The interval is divided in two parts, recursively, until a given threshold

Table 5.3: Number of spawned processes per node and the execution time (in seconds) to search prime numbers in an interval from 1 until 20 million.

	Node 1	Node 2	Node 3	Node 4	Node 5	Time (sec)
LAM/MPI	39	0	0	0	0	181.15
Scheduling library	8	8	8	8	7	46.12

is reached. Then, the prime numbers are searched in the sub-intervals, those found are sent to the upper level, which merges the two children results. When the top of the recursion is reached, there are the primes numbers between 1 to N. The MPI-2 implementation replaces the recursive calls to dynamic process creation.

Table 5.3 shows the process mapping as well as the execution time of MPI prime numbers application. The interval analysed is from 1 to 20 million with a threshold equal to 1 million. As expected, the execution time shows that the application performance was improved taking advantage of a balanced process distribution among the available nodes. However, in some cases, provides only a balanced distribution is not enough to provide high performance. For instance, the computation cannot be egalitarian partitioned among the units of execution to irregular applications. The search for prime numbers is an example of irregular application: the incidence of prime's numbers varies from interval to interval, thus the time spent searching, and sending and merging them is different to each process. Aiming to improve the execution of this kind of application, we proposed a workload-based strategy as described in the following.

5.2.3.2 ***Workload-based Mapping***

The goal of the workload-based strategy is to identify what is the node with the lowest workload, which will be the best candidate to receive the spawning MPI process. We proposed this strategy in CERA et al. (2006). The current workload is determined collecting usage information from monitoring tools of the Operation System running on the available nodes. Thus, we included two new components in the scheduling library to deal with the monitoring issues.

One of them deals with the collection, storage and analysis of the usage information. It was called *load monitor* and it is a daemon that is launched at starting time, one to each used node of the application. It retrieves usage metrics (such as CPU, memory, network usage, etc), which are stored in a LRU (Least Recently Used) buffer, so the oldest values are thrown away, and the buffer reflects the current load of nodes. The workload of a node is the average of the values in the buffer of its load monitor in a given moment. This mechanism, known as Single Moving Average, is used on Time Series Forecast Models based on the assumption that the oldest values tend to a normal distribution (HAMILTON, 1994). The average also smoothes the effect of floating variations of the load, that does not characterize the current usage of nodes.

The other component is responsible to manage the load monitor daemons and centralize their workload information. This component is named *resource manager - RM*, because it maintains the nodes ordered according to their usage from the lowest to the largest workload. Thus, to every new dynamic MPI processes, the scheduler daemon requests the RM, which responds the first node of its list. Aiming to minimize the impact of the communications between RM and the load monitors

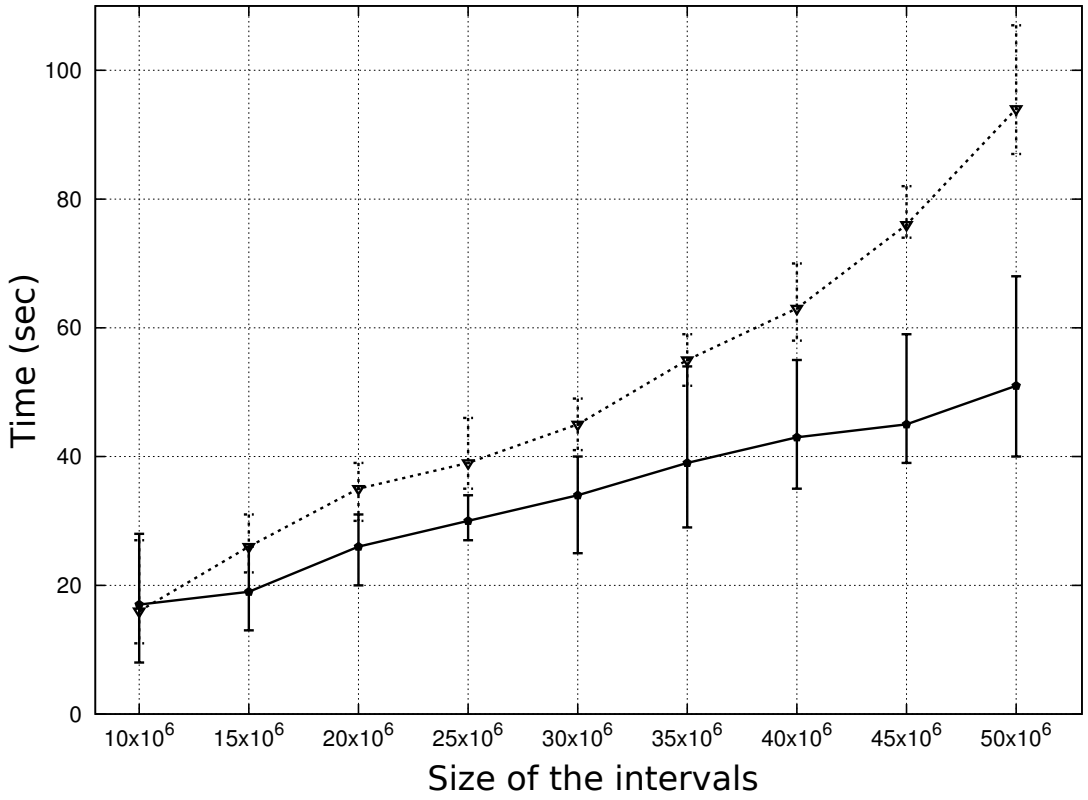


Figure 5.10: Comparison between Round Robin (dotted line) and workload-based (solid line) strategies: The execution time of prime numbers application varying the size of the intervals.

and avoid bottlenecks, the RM chooses randomly different intervals to query each load monitor. Furthermore, the RM was planned including an interface that allows using third-party resource managers or monitors. This can be useful for interaction with grid middlewares like Globus (FOSTER, 2006).

To analyse the performance of the workload-based strategy, we collected the CPU usage during the execution of the search for prime number previously described. The nodes with the lowest CPU usage will be the best candidates to receive the dynamically spawned processes. Figure 5.10 shows a comparison of the execution time achieved with Round Robin and workload-based strategies, varying the size of intervals in which the prime numbers are searched. Regarding the graph, we can see the impact of taking into account on-line workload information to an irregular application. The workload-based policy enables that under-loaded processors execute more processes than over-loaded ones improving the application performance.

5.2.4 The Dynamic Process Scheduler supporting Malleability

In Section 5.2.2, we showed the support provided by the LAM/MPI and Open-MPI distributions to manage volatile processors. However, we identified that there are some issues to be externally provided to execute malleable MPI applications:

- **LAM/MPI:** `lamgrow` and `lamshrink` commands deal with volatile processors at runtime, but the changes are not automatically perceived by the MPI applications. Furthermore, these commands require the identification of the nodes involved, such as the host name of nodes or their node identifier;

- **OpenMPI:** although it does not provide a native support to volatile processors, this can be implemented since some constraints are taken into account, such as to start the application including nodes that will be added at runtime, and do not kill the `orted` daemon on released nodes. In OpenMPI, it is also required to inform the application about changes in the processors availability and identify the nodes involved in changes.

Thus, we take advantage of our previous work, the dynamic process scheduler shown in Section 5.2.3, to provide these external information allowing the management of the volatile processors as well as making applications aware of the changes. Using the scheduling library we are able to manage (and know where are) the dynamic MPI processes in two moments: at starting time when the distribution happens on static processors; and after changes in the set of processors.

The information about the processors availability comes from RMS systems, in our case, OAR. Thus, we updated the scheduler to allow interactions with OAR. The ways in which the communication happens and further issues, will be exposed in Section 5.2.5. In this section we will explain how the information is treated and what are the reactions caused by them. Thus, for the moment, it is enough to consider that the information is available. In the following, Section 5.2.4.1 exposes the technical issues of the LAM/MPI distribution, and Section 5.2.4.2 shows the OpenMPI ones.

5.2.4.1 ***Technical Issues using LAM/MPI***

The scheduling library behavior described in Section 5.2.3 stayed the same for this version that aims at malleability: a scheduler daemon runs along with the user application; and every call to `MPI_Comm_spawn` is overloaded allowing querying the scheduler daemon about the best physical location to new processes. In the following we will describe the added features to deal with volatile processors as well as how these features are implemented using LAM/MPI.

At starting time. On static environments, as those targeted in the first version of the scheduling library, it was not required to keep a list of used nodes in the scheduler daemon. However, using volatile processors, keeping this information updated is important to guide the adaptive actions. Thus, at starting time the scheduler fills a data structure storing the identification of each available node, which will be updated whenever any change happens (as described in the following items). The host names of the nodes are caught from the LAM/MPI internal data structures. Furthermore, the scheduler also catches the number of cores and counts the number of processes of each node. This is helpful in multi-core environments to guide the process mapping: the scheduler tries to keep each node with as many processes as cores;

When OAR provides further processors. When the scheduler daemon receives a notification from OAR announcing new nodes, it identifies the host name and executes a system call to run the `lamgrow` command to each new node. Thus, the new nodes become part of the LAM/MPI network. Furthermore, these nodes are inserted in the scheduler list. This ensures that they will be taken into account during future requests of process mapping;

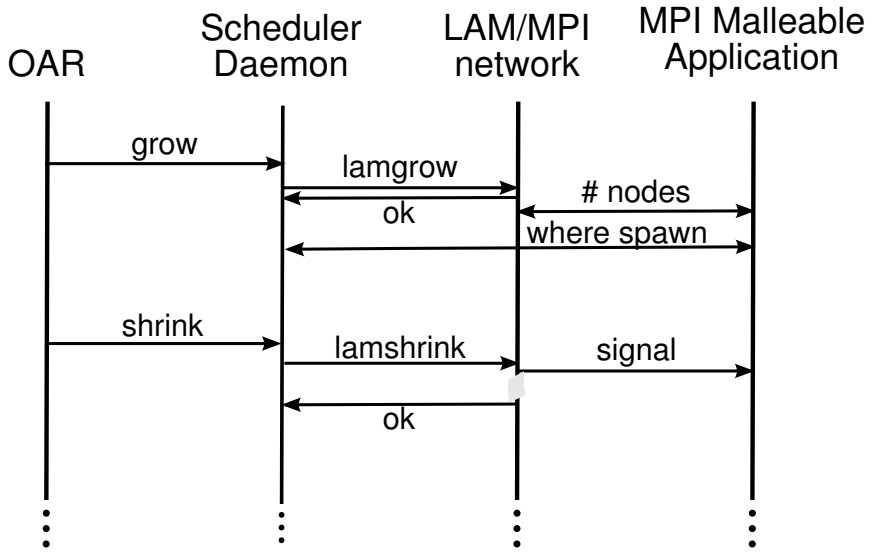


Figure 5.11: Changing the current number of nodes through LAM/MPI: *(i)* to increase - OAR announces new nodes (*grow* arrow), the scheduler calls `lamgrow`, the application identifies the increase and performs the growth spawning new processes into the new nodes; *(ii)* to decrease - OAR request some nodes (*shrink* arrow), the scheduler calls `lamshrink`, warns the running processes about the request, and the warned processes finalize.

When OAR requires processors. When OAR requests some nodes, it can provide two information: *(i)* the number of requested nodes meaning that any nodes can be released since the number is reached; or *(ii)* a list of host names identifying those that must be released. For the first kind of information, the scheduler daemon will release nodes from the tail of its list of used nodes. For the second, the scheduler searches for the exact host names in the list. In both cases, the node identifier of the required nodes is set as parameter of the `lamshrink` that is executed as a system call. Thus, the LAM/MPI network releases the required processors, and they are deleted from the scheduler list.

Notice that the actions described above update the LAM/MPI network and the scheduler daemon. However, the running application must be able to identify changes in order to react. Our malleable MPI applications constantly query the current number of nodes to LAM/MPI (using `getntype()`). When the number is increased, the applications execute the growth action: as many processes are spawned as there are further nodes. To each `MPI_Comm_spawn` call, the scheduler daemon ensures that these new processes are placed in further nodes. When the number is decreased, the applications execute the shrinkage action: processes running on required processors are warned, they inform which tasks are being executed on them that must be recomputed, and finalize. To avoid crashes, the remaining processes must also know about the releasing ones to update the waiting parameter and wait no more for finished processes.

Figure 5.11 illustrates the reactions to the inclusion and releasing of nodes in malleable MPI application at runtime. The *grow* arrow represents OAR announcing further nodes to the scheduler daemon. The scheduler calls `lamgrow` to add the nodes on the LAM/MPI network, when the command ends the LAM/MPI network is updated (*ok* arrow), *i.e.* each further node runs a `lamd` daemon. Then, in one of

its periodical requests (*# nodes* arrow), the MPI application identifies the increase in the current number of nodes. It reacts calling the spawn and the scheduler ensures that the dynamic processes will be placed into the further nodes (*where spawn* arrow).

The *shrink* arrow represents OAR requesting some nodes of the malleable MPI application. The scheduler daemon receives the request and identifies which nodes are required. Then, it calls the `lamshrink` command to release them from the LAM/MPI network. This command warns the running processes of the requested nodes about the impending release by a signal (*signal* arrow). The `lamshrink` waits for a certain time while the warned processes finish. The return of the `lamshrink` means that the nodes were released, *i.e.* their `lamd` daemons were killed.

5.2.4.2 ***Technical Issues using OpenMPI***

Similarly to the previous section that described the technical issues of LAM/MPI in the adaptation of the scheduling library to deal with malleability, this section is focused on OpenMPI technical issues:

At starting time. The scheduler daemon keeps a list with all nodes that can be used by the MPI application. This list is filled using the host names file passed as parameter to the `mpirun` or `mpiexec`. However, notice that to allow the growth at runtime, this file includes the current available nodes as well as those that can be added at runtime (as explained in Section 5.2.1.2). Thus, the scheduler is able to identify which are currently available to ensure that only they will receive dynamic processes during the malleable application runtime. In other words, the management of the list of nodes allows the scheduler daemon to control the utilization of the nodes of the OpenMPI network;

When OAR provides further processors. In OpenMPI, the nodes that can be added on-line are figuring in the scheduler list of nodes since the application startup, but set as not available. When OAR announces further nodes, the scheduler changes their status to available and then they will be taken into account in future process mapping;

When OAR requires processors. The scheduler receives a request that can inform: *(i)* the number of required nodes; or *(ii)* the host name of the required nodes. Similarly to the LAM/MPI version, the scheduler either will release the last nodes in its list until reaches the required number, or will search for the specific host names. However, as OpenMPI do not offer a mechanism to release nodes like the `lamshrink`, the scheduler implements such a mechanism itself. Knowing which nodes must be released, the scheduler sends a signal to their running processes warning about the release. Notice that the `orted` daemon will keep running on released nodes as the OpenMPI restriction.

Again, malleable MPI applications also must discover the changes in the OpenMPI network allowing their reaction. As OpenMPI is not able to answer the current number of nodes, applications query the scheduler daemon to retrieve this information. When changes are detected, the application executes the coherent action: growth or shrinkage. The instructions set of these actions is exactly the same described to LAM/MPI.

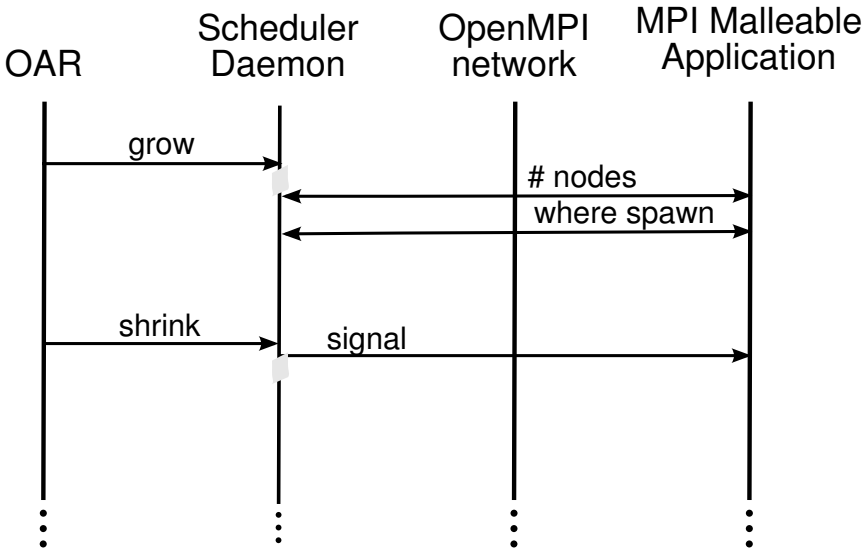


Figure 5.12: Changing the current number of nodes through OpenMPI: *(i)* to increase - OAR announces new nodes (*grow* arrow), the scheduler sets the new nodes as available, the application identifies the increase and performs the growth spawning new processes into the new nodes; *(ii)* to decrease - OAR request some nodes (*shrink* arrow), the scheduler identifies the requested nodes, warns their running processes about the request, which finalize.

Figure 5.12 illustrates the actions to increase and decrease the current number of nodes according to OAR decisions. When OAR announces further nodes (*grow* arrow), the scheduler searches for them in its list of nodes and updates their state to available. Then, when the MPI application identifies the increase (through the request of the current number of nodes – *# nodes* arrow), it performs the growth action spawning dynamic processes into the new processors thanks to the scheduler. When OAR requests some nodes (*shrink* arrow), the scheduler identifies which are requested and warns their running processes through a signal. When the application catches the signal, it performs the shrinkage action informing which tasks were being computed and finalizing the execution through `MPI_Finalize`.

5.2.5 Interactions between OAR and Dynamic Process Scheduler

Section 5.1.2 presented how OAR decides about the availability of the processors. In addition, Section 5.2.4 shows the actions performed in the application side to provide adaptation as the changes are announced. The bridge between the decision in the OAR side and the reactions in the application side will be introduced in this section. The interactions between OAR and the dynamic process scheduler are responsible by the success of malleable MPI applications executions. Figure 5.13 illustrates the combination of OAR decisions and the application adaptability.

As we saw in Section 5.2.4, the information required by the malleable applications from OAR are:

- The host names of available nodes at malleable MPI application starting time;
- The host name of the nodes being added at runtime;
- The number of nodes or their host names when they are required at runtime.

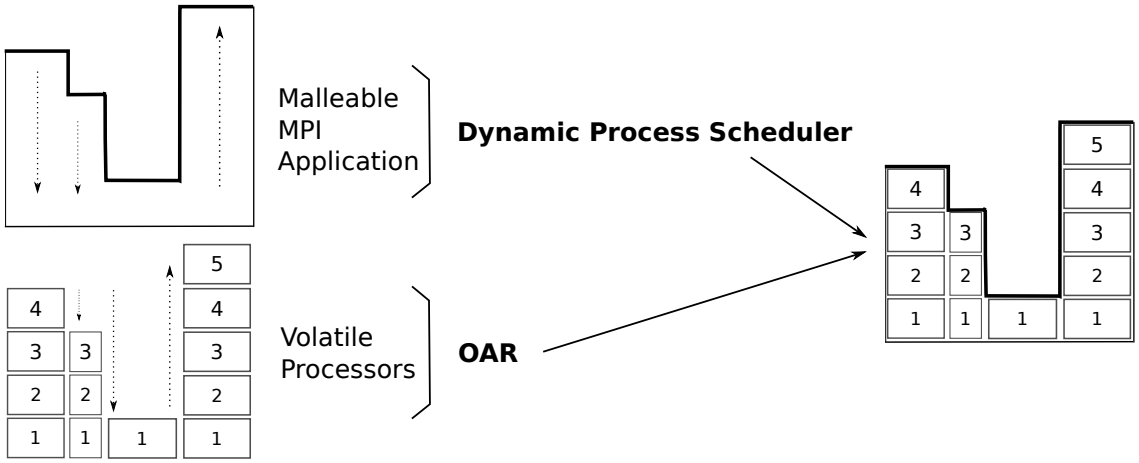


Figure 5.13: Requirements of malleability: OAR resource manager and the dynamic process scheduler interacts changing information about the processors availability.

The mechanism chosen to implement the information exchanging between OAR and the scheduler daemon was through files. In a traditional usage of OAR, it provides a file with the host names of the nodes allocated to a job, which it is accessible by an environment variable – `$OAR_NODEFILE`. To malleable jobs, the host name of starting nodes are also stored in this predefined file, satisfying the first item above. In addition, we included two other files to the remaining items. The `$OAR_GROWFILE` has the host names of the further available nodes and the `$OAR_SHRINKFILE` can have either a number that represents how many nodes are required, or the host name of the required nodes.

Once OAR provides files with the required information, the dynamic process scheduler can read such information and then launch the coherent adaptive action. To ensure that an addition or exclusion of nodes will only happen once, the scheduler deletes the files every time that they are read. We implemented a mechanism that to every write done by OAR either in `$OAR_GROWFILE` or `$OAR_SHRINKFILE`, the scheduler is warned and proceed with the coherent treatment of the information. This mechanism is transparently implemented through bash scripts and reduces the delay between the OAR write and the scheduler read.

Notice that the solution previously described meets the goals of our first effort to provide malleability in cluster environments managed by OAR. However, considering the goals of a production support of malleability, maybe this simple mechanism to exchange information through files will not be efficient. The OAR team is investigating ways to improve this mechanism. Anyway, this simple solution supplies our current demand that aims to test the execution of malleable MPI applications in a cluster environment.

5.3 Execution of Malleable Jobs in a Cluster Environment

This section exposes results of our experiments with OAR supporting malleable MPI applications. The cluster environment used in our tests is part of the Grid5000 platform (BOLZE et al., 2006). We used a cluster (IBM System x3455) composed by DualCPU-DualCore AMD Opteron 2218 (2.6 GHz/2 MB L2 cache/800 MHz) and 4GB memory per node with Gigabit Ethernet. We used our dynamic process sched-

uler with LAM/MPI distribution version 7.1.4. We opt by LAM/MPI because it was the most stable when we perform the executions (in 2008 to 2009). Furthermore, the content of this section was published in CERA et al. (2010).

We implemented a malleable MPI application that computes the Mandelbrot fractal (described in Section 4.1.4). We implement the procedures described in Section 4.2.2.2 to provide a Master/Worker malleable MPI application. The master identify changes in processors availability through a mechanism similar to the `malleability_handle` (see Figure 4.11). The growth action means to spawn new workers in recently added processors, and int the shrinkage action, the worker catch a signal that causes the stopping of current task computation, a send of a message informing which is the current task, and the worker finalization.

Two series of experiments were performed. First, the adaptive actions were tested without the intervention of OAR. Thus, we can evaluate the impact of malleability in application performance without considering the issues related to the resource management. These results are presented in the Section 5.3.1.

The second set of experiments evaluates the improvements of resource utilization achieved using malleable jobs. Aiming to have a cluster loaded with rigid jobs in such a way that we were able to repeat the experiments using different configuration, we took workload traces of a production clusters. They are used to load a cluster of the Grid5000 through OAR, in a mechanism of workload trace injection. The used traces comes from the DAS2 grid platform (LI; GROEP; WOLTER, 2004), which represent the workload of twelve months (year 2003) on 5 clusters. To provide the results exposed in Section 5.3.2 we selected a specific cluster and a time range of the DAS2 traces. Along with this injection mechanism, we have a demo that submits malleable jobs which run on free processors of the cluster (*i.e.* those not used by the rigid jobs). As a restriction of the OAR system, we will have one malleable job at a time.

5.3.1 Performance of Malleable MPI Applications

In the experiments of this section, we used at most 64 cores from 16 nodes of a Grid5000 cluster (2 CPUs with 2 cores per node) and with one MPI process on each available core. Figures 5.14 and 5.15 show the execution time of the malleable MPI application when it grows and shrinks, respectively. For Figure 5.14, the x axis values are the initial number of cores (number of cores at starting time = 16, 32, or 48 cores), which represent 25%, 50% and 75% of the 64 cores available in the cluster. At runtime, the number of cores is updated until achieve the 64 ones. On the other hand, in Figure 5.15 the execution starts with 64 cores and is updated until the values showed in the x axis (16, 32, or 48 cores). Both graphs represent the average time of multiple executions to each configuration, in such a way that the standard deviation is around of 1.2.

Furthermore, we controlled the intervals in which the growth and shrinkage actions happen in two ways: *(i)* providing a dynamic event **at** a time limit, and *(ii)* providing gradually new dynamic events **until** a time limit. Time limit is defined as 25%, 50% or 75% of the parallel reference time (graphs include dotted lines pointing these percentages). Reference time is always the execution time achieved with the initial number of cores, *i.e.*, x cores to growth and 64 cores to shrinkage.

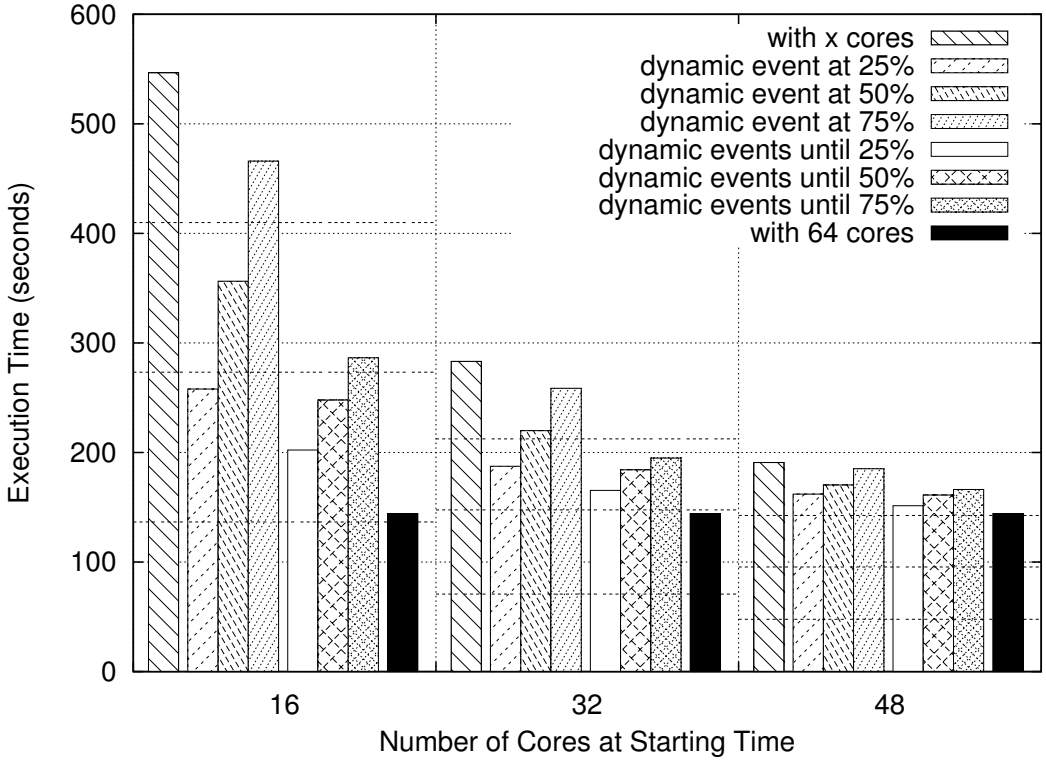


Figure 5.14: Execution time of the malleable MPI application when it grows: at starting time the application has as many processes as the number of cores showed in the horizontal axis (16, 32, or 48 cores), and it grows until 64. Dynamic events happen at or until a time limit.

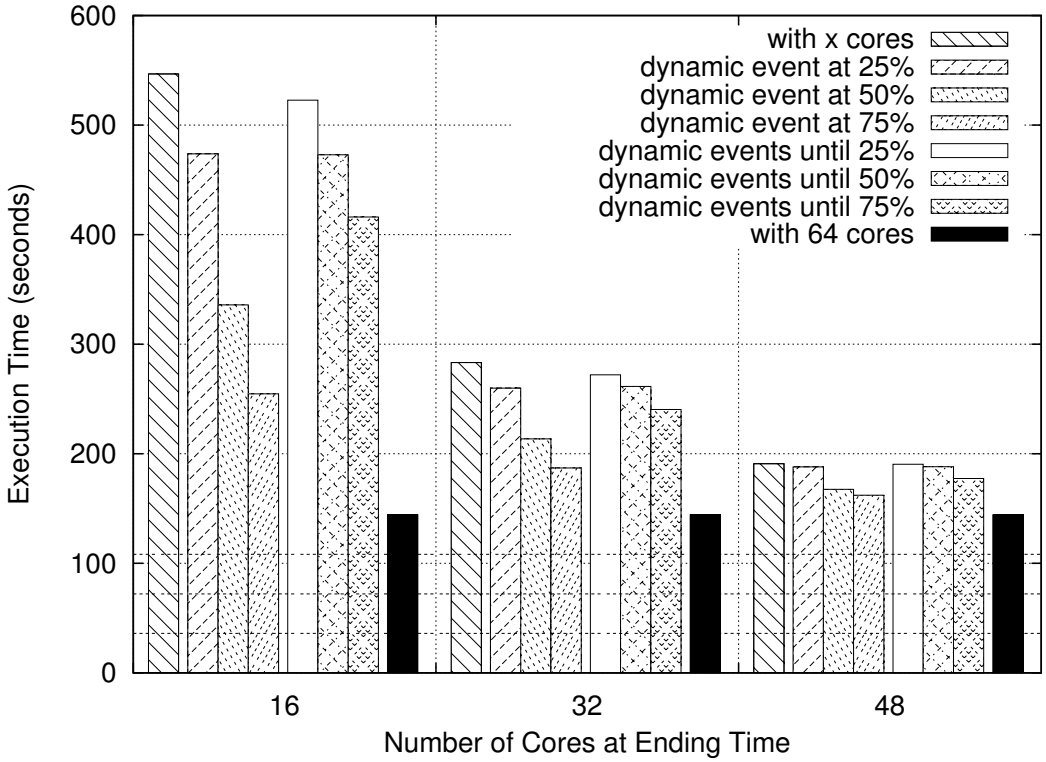


Figure 5.15: Execution time of the malleable MPI application when it shrinks: at starting time the application has 64 processes and shrinks until as many processes as the x values (16, 32, or 48 cores). Again, dynamic events happen at or until a time limit.

In Figure 5.14, the first bar of each x value represents the reference time, *i.e.* the execution time using x cores. Notice that, the reference time is greatest one (the greatest bar), which means that the addition of new cores at runtime always improve the application performance. The execution time with 64 cores is the last bar to each x value, which represents the best case, *i.e.* when the application performs with as many cores as possible. Moreover, regarding the results we can took two other conclusions: *(i)* gradual addition of nodes ("dynamic events until") is more efficient than batch addition ("dynamic event at") because the application is able to use the new cores as soon as they are added. This can be seen comparing the three bars of "dynamic event at" (second, third, and fourth bars) with the three of "dynamic events until" (fifth, sixth, and seventh bars); *(ii)* as soon as extra nodes are available, greater is the improvement caused by them. To all initial numbers of cores, the best performance is always achieved when the extra nodes are added until 25% of the reference time.

Figure 5.15 has the same organization of Figure 5.14: the first bar of each x is the execution time with x cores and the last with 64. However, in these tests the application starts using as many cores as possible (64 ones) and releases them at runtime. Thus, the reference time is with 64 cores. We can observe that the time when the application shrinks is always greater them the reference time, but never greater than the time with x cores. This means that although the application loss nodes at runtime, while they are available, they are able to provide some gains. Notice that our solution does not avoid re-computations, because tasks running on released nodes are restarted, and yet it is able to provide gains. As expected, the shrinkage achieved the opposite behavior than growth: *(i)* the best gains are achieved when the nodes are released at a time limit (comparing the "dynamic event at" bars with "dynamic events until" ones); and *(ii)* as longer it takes with nodes, the best is the application performance. To all scenarios, the best performance is when the dynamic events happen at 75% of the reference time.

In the following, we analyse the speedup achieved when the malleable application grows and shrinks. Let p_r be the number of cores on which a reference time t_r has been measured. Thus, the application performs $W = t_r \times p_r$ operations during reference time collection. Then, the application progressively changes the number of cores from p_r to p:

- In the first experiment (dynamic events performed at a time limit), it runs during $\alpha t_r (0 \leq \alpha \leq 1)$ seconds on p_r cores, and then on p ones;
- In the second experiment (dynamic events performed until a time limit), it runs during $\alpha t_r (0 \leq \alpha \leq 1)$ seconds on a gradual increasing or decreasing number of cores, until p cores. Then it continues the execution until completion, without any change. The number of cores changes at regular timesteps in c units. In fact, $c = 4$, since the `lamgrow` command allows to add a whole node (2 CPUs with 2 cores) in LAM/MPI network. The number of gradual timesteps is therefore $\frac{p-p_r}{c}$, and each one lasts $\delta = \frac{\alpha t_r}{\frac{p-p_r}{c}}$ seconds.

The ideal speedup in the first experiment. The execution on p_r cores has duration of αt_r. At this point, there are $(1-\alpha)W$ operations to be performed by p cores. Ideally, this second phase runs in $(1-\alpha)W/p$ time. Therefore, the total parallel time in this case is $t_p = \alpha t_r + (1-\alpha)t_r p_r/p$, and the speedup t_r/t_p is:

$$S = \frac{1}{\frac{p_r}{p}(1-\alpha)+\alpha} \tag{5.1}$$

The ideal speedup in the second experiment. As in the previous case, the number of operations performed during the first αt_r seconds, can be computed to obtain the parallel time of the second part of the execution on p cores. At the ith timestep ($i = 0...\frac{p-p_r}{c}-1$), the program is running with $p_r + ci$ cores in δ seconds. Therefore, the number of operations n_i that were executed is $\delta(p_r+ci)$. When all the p cores are available (*i.e.* at αt_r time), $\sum_{i=0}^{(p-p_r)/c-1} n_i$ operations have been run, leaving $W - \sum_{i=0}^{(p-p_r)/c-1} n_i$ to be run by p cores. Thus, the second phase has duration:

$$\frac{t_r p_r - \sum_{i=0}^{(p-p_r)/c-1} \delta(p_r+ci)}{p}, \tag{5.2}$$

and the total execution time in this second experiment is therefore:

$$t_p = \alpha t_r + \frac{t_r p_r - \sum_{i=0}^{(p-p_r)/c-1} \delta(p_r+ci)}{p}. \tag{5.3}$$

Besides,

$$\sum_{i=0}^{(p-p_r)/c-1} \delta(p_r+ci) = \delta p_r \frac{p-p_r}{c} + c\delta \sum_{i=0}^{\frac{p-p_r}{c}-1} i = \delta p_r \frac{p-p_r}{c} + c\delta \frac{(\frac{p-p_r}{c}-1)\frac{p-p_r}{c}}{2}. \tag{5.4}$$

And since $\delta = \frac{\alpha t_r}{\frac{p-p_r}{c}}$, the later equation yields:

$$\sum_{i=0}^{(p-p_r)/c-1} \delta(p_r+ci) = \alpha t_r p_r + \frac{\alpha t_r}{2}(p-p_r-c). \tag{5.5}$$

Therefore, the total parallel time in this case is:

$$t_p = \alpha t_r + \frac{t_r p_r - \alpha p_r t_r - \frac{\alpha t_r}{2}(p-p_r-c)}{p} = \frac{t_r}{2p}(2p_r + \alpha(p-p_r+c)). \tag{5.6}$$

And the parallel speedup t_r/t_p is:

$$S = \frac{2p}{2p_r + \alpha(p-p_r+c)}. \tag{5.7}$$

Table 5.4 shows the speedups of the malleable MPI application when it grows (upper table) and shrinks (lower table). The first and second columns show the number of used cores before and after the changes, respectively. The third column represents the ratio of the time in which the changes happen. The fourth and sixth columns are the speedup in practice (t_r/t_p) for the first and second experiments, respectively. Finally, the fifth and seventh columns are the ideal speedups of the first (as Equation 5.1) and second (as Equation 5.7) experiments.

Table 5.4: Speedup of the malleable MPI application when it grows (upper table) and shrinks (lower table): p_r - cores at starting time; p - cores after the change; α - ratio of reference time in which changes are performed ; S - speedup in practice t_r/t_p to the first (fourth column) and second (sixth column) experiments; $S(Eq.5.1)$ - ideal speedup of the first experiment as Equation 5.1; and $S(Eq.5.7)$ - ideal speedup of the second experiment as Equation 5.7.

Speedup with Growth						
			Adding AT		Adding UNTIL	
p_r	p	α	S	$S(Eq.\ 5.1)$	S	$S(Eq.\ 5.7)$
16	64	0.25	2.12	2.29	2.70	2.84
		0.50	1.53	1.60	2.20	2.21
		0.75	1.17	1.23	1.91	1.80
32	64	0.25	1.51	1.60	1.71	1.75
		0.50	1.29	1.33	1.54	1.56
		0.75	1.09	1.14	1.45	1.41
48	64	0.25	1.18	1.23	1.26	1.27
		0.50	1.12	1.14	1.18	1.21
		0.75	1.03	1.07	1.15	1.15
Speedup with Shrinkage						
			shrinking AT		shrinking UNTIL	
p_r	p	α	S	$S(eq.\ 5.1)$	S	$S(eq.\ 5.7)$
64	16	0.25	0.30	0.31	0.28	0.27
		0.50	0.43	0.40	0.30	0.30
		0.75	0.57	0.57	0.35	0.34
64	32	0.25	0.55	0.57	0.53	0.53
		0.50	0.68	0.67	0.55	0.56
		0.75	0.77	0.80	0.60	0.60
64	48	0.25	0.77	0.80	0.76	0.77
		0.50	0.86	0.86	0.77	0.79
		0.75	0.89	0.92	0.81	0.81

In the speedups when the application grows (upper table), practical speedups are slightly lower than the ideal ones, representing the overhead to perform the spawning of new processes. According to LEPÈRE; TRYSTRAM; WOEGINGER (2002) the standard behavior in on-the-fly resources addition is that such addition cannot causes an increasing in overall execution time of the application. In Table 5.4, we observe that the speedups with growth operations are always greater than 1, meaning that the addition of new cores could improve application performance as expected. In the speedups when the application shrinks (lower table), all speedups values are lower than 1, because the exclusion of nodes decreases the performance. In this case, practical speedup is quite similar to the theoretical and their variations are due to the execution of shrinkage procedure: interrupt the task execution; inform the current task and finalize the workers processes.

5.3.2 Analysis of the Cluster Utilization using Malleability

The goal of the experiments showed in this section is to evaluate the impact of malleable MPI applications on cluster utilization. A workload with a slice time of 5 hours of DAS2 platform with 40% of cluster utilization was injected into OAR. This workload loads the resources, representing the normal workload of the cluster. Along with the rigid jobs, a malleable job is submitted and runs upon the free resources, *i.e.* which are not used by the normal workload.

The jobs that compose the normal cluster workload (from DAS2 workload traces) are empty jobs, *i.e.* processors allocated only to run `sleep` commands for a specific time. Since the malleability is performed using whole nodes, there was no need to perform real application in the jobs of the normal workload. As explained in the Section 5.1.3, the OAR malleable job has two parts: the rigid part using only one node (minimum unit manageable by LAM/MPI); and the *Best Effort* part that is as large as the number of free processors less one (that runs the rigid part).

The execution of our malleable application is compared with a moldable-*Best Effort* one. A moldable-*Best Effort* job is defined as a moldable job that can be executed upon all the free processors, but it will be immediately killed, like a *Best Effort* job, when arriving jobs request some of its processors. Thus, we change the demo that launches malleable jobs on the unused processors to submit moldable-*Best Effort* jobs. Thus, both jobs start in the same conditions allowing the comparison.

Figures 5.16 and 5.17 show the results of malleable and moldable-*Best Effort* jobs, respectively. The white portions represent the time in which processors remaining unused. Thus, it can be observed a gain in cluster utilization with malleable jobs. Furthermore, the white vertical lines of figures also represent the time between the finalization of a job and the starting of the another. Additionally, OAR does not assign processors that will be soon required by rigid jobs. The resource discovery command operates with its default time to look forward in reservations (near-future rigid jobs of the normal workload).

In terms of cluster utilization, the normal workload uses 40% of the cluster, thus, there are 60% unused. Moldable-*Best Effort* jobs used 32% of the free processors, arriving at 72% of overall cluster utilization. On the other hand, malleable jobs used 58% of the free processors, arriving at 98% of overall cluster utilization. Hence, an improvement greater than **25%** in cluster utilization is achieved when our malleable approach is compared with a non-dynamic one.

Furthermore, we count the number of jobs executed during the 5 hours of experimentation. With the malleable jobs, we obtained 11 successfully 'Terminated' malleable jobs, compared to 3 'Terminated' and 7 in 'Error State' for the moldable-*Best Effort*. The impact of the response time, for the normal workload, was also measured. The results have shown 8 seconds of average response time to moldable-*Best Effort*, compared to 44 seconds to malleable. These values are explained by the OAR grace time delay (*i.e.* the time spent to wait the processes finalization in shrinkage actions) that is 40 seconds for malleable jobs.

Although our initial experiments consider only one workload trace and one malleable job at a time, they show the potential of malleable jobs as a tool to improve cluster utilization. In other words, we achieved our experiments goal, which was to verify if the efforts to enable malleable jobs on OAR would be outperformed by the gain reached in processors utilization.

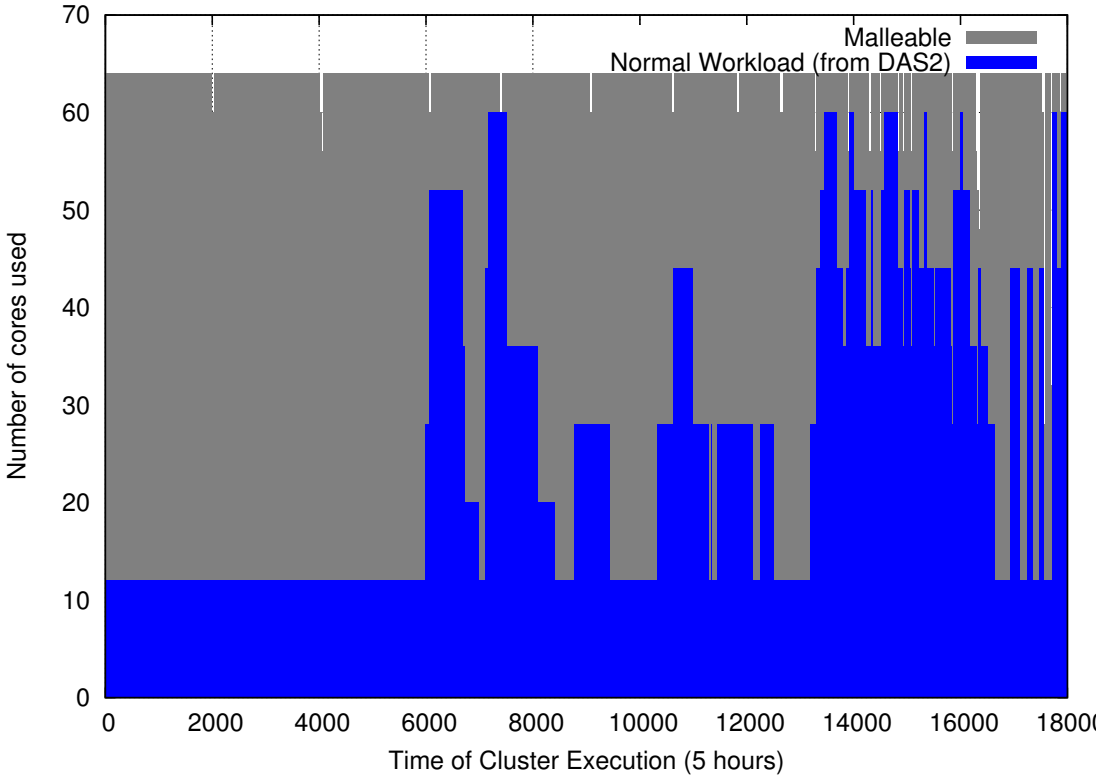


Figure 5.16: Malleable jobs performing on free processors of a cluster: the normal workload uses 40% of the cluster while the malleable jobs use 58% of the 60% remaining.

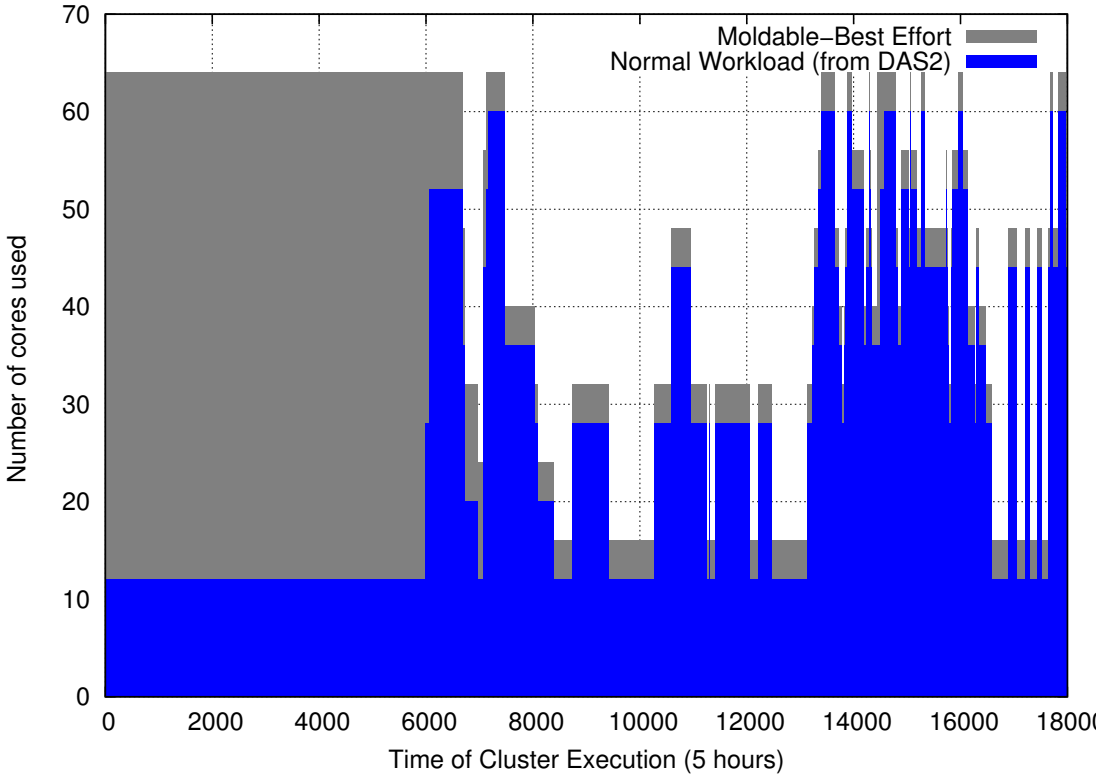


Figure 5.17: Moldable-*Best Effort* jobs performing on free processors of a cluster: the normal workload uses 40% of the cluster while the moldable-*Best Effort* use 32% of the 60% remaining.

5.4 Conclusion

This chapter discussed the main required issues to execute malleable MPI applications in cluster environments. In this sense, we presented the OAR resource manager (Section 5.1) that has a scheduling policy able to manage volatile processors: it identifies the unused processors, providing them to malleable applications, and managing their availability taking into account the requirements of the rigid jobs (standard utilization of the cluster). Furthermore, OAR supports malleable jobs, *i.e.* jobs that change their size at runtime, taking advantage of its *Best Effort* jobs. Thus, a malleable job in OAR is composed by two parts: *(i)* a rigid part representing the invariable part of the job; and *(ii)* a *Best Effort* one that may change its size at runtime offering some flexibility to the job.

In the following, we showed the main aspects required to allow MPI applications to deal with volatile processors (Section 5.2). In other words, we showed the technical issues in the development and execution of malleable MPI applications. Thus, we presented the MPI distributions and which support they offer to malleability: allowing dynamic process creation and means to deal with volatile processors at runtime. Furthermore, the mapping of dynamic processes is a key issue in malleability. Then, we showed the default process mapping offered by MPI distributions, as well as their deficiencies. We introduced our scheduler that map dynamic MPI processes with two mapping strategies: the Round Robin and a workload-based. This scheduler was extended to supply the requirements of malleability. In this sense, the scheduler goal is to balance the distribution of the processes taking into account the current availability of the processors. Finally, to know about changes in the set of available processors, our scheduler interacts with OAR. Thus, it is able to inform the application when changes happen, and launch the required adaptations.

Due to an international cooperation with the LIG laboratory in France, we were able to implement the integration of OAR with our scheduler, allowing the execution of malleable MPI applications in cluster environments. Section 5.3 exposed our experimental results executing a malleable MPI application through OAR. It includes an analysis of the application performance able to grow and shrink as the availability of processors and the improvements in cluster utilization achieved executing malleable jobs along with the normal workload (standard rigid jobs).

We concluded that the complexity to support malleability is outperformed by achieved gains. This conclusion is based on the experiments that showed a gain up to 25% in cluster utilization using malleable jobs when compared to moldable-*Best Effort* ones. We mesured it using real workloads traces to charge our cluster testbed processors through OAR. Thus, the contribution of this chapter is a description of the challenges to support malleability in resource manager (proper scheduling, flexible allocation of the processors, etc) as well as in the MPI context (support to dynamic process creation, the mapping of these dynamic processes, the launching of adaptive actions, etc).

Once we identified that dynamic process creation can be used to develop malleable MPI applications, we will focus the next chapter on issues of evolving MPI applications. As previously mentioned, the evolving applications aimed in our work are those follow the explicit task parallelism programming paradigm and we investigate the use of the MPI-2 features to help to unfold the parallelism at runtime.

6 EXPLICIT TASK PARALLELISM ON MPI APPLICATIONS

This chapter aims to provide explicit task parallelism on MPI applications taking advantage of the MPI-2 features, in special the dynamic process creation. These applications can be seen as a sub-class of evolving ones, because they are able to exploit the potential parallelism at runtime, adapting themselves according to the target architecture and input data. In Chapter 4, we introduced ways to develop explicit task parallelism in MPI applications, their requirements and an example in which a problem is solved based on this programming paradigm. We could observe that there are many challenges to provide this programming paradigm on a distributed-memory context. This chapter discusses about their main issues:

- **Definition of the abstract MPI tasks:** how to allow the dynamic unfolding of the parallelism in MPI programs; and how to determine the granularity of the abstract tasks to achieve efficiency – Section 6.1;
- **Solving dependencies among abstract MPI tasks:** how to take advantage of the MPI communication features to provide high performance data transfers among abstract MPI tasks – Section 6.2;.
- **Scheduling abstract MPI tasks at runtime:** how to provide on-line decisions to get load balancing – Section 6.3.

In Section 6.4, we present our first experiments executing MPI applications following the explicit task parallelism paradigm in a cluster environment. Additionally, Section 6.4.5 presents our reflexions about the adaptation of the explicit task parallelism into the MPI standard. Finally, Section 6.5 exposes the final considerations of this chapter.

6.1 Defining Abstract MPI Tasks

Explicit task parallelism becomes popular as a powerful programming paradigm to multi-core architectures, because of its efficient and elegant way to extract the potential parallelism of the algorithms. It specifies units of work (abstract tasks) that can be executed in parallel as well as their dependencies inside of the irregular programs, leaving it up to the runtime to unfold the parallelism and schedule the dynamic generated tasks. Thus it decides if tasks will be executed on different units of execution (processes or threads), or simply run sequentially.

```
int fib( int n ){
  if( n < 2 )
    return n;
  else{
    int x, y;
    x = fib( n-1 );
    y = fib( n-2 );
    return( x + y );
  }
}
```

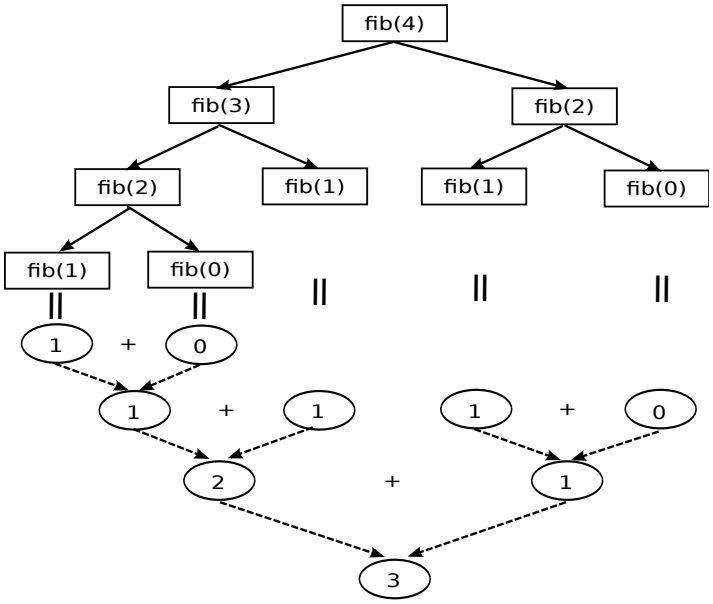


Figure 6.1: Unfolding of the Fibonacci execution: the recursive calls of `fib( n-1 )` and `fib( n-2 )` can be performed in parallel as illustrated in the right; when the results are available, the dependencies are satisfied.

Explicit task parallelism rises in Fork/Join programs (saw in Section 2.5.4): *fork* – tasks are generated at runtime while there is parallelism to be explored (*i.e.*, as the input data and algorithm features); and *join* – executed tasks join to compose the application results. As an example, we consider the Fibonacci computation (previously introduced) to illustrate the Fork/Join program structure (see Figure 6.1).

In the left of Figure 6.1, there is the sequential C function to search for the *ith* element in Fibonacci (`n`), which performs recursive calls searching for `n-1` and `n-2`. According to the explicit task paradigm, this function can be seen as an abstract task, which is dependent of the input – the argument `n`. Moreover, the recursive calls represent the unfolding of the parallelism: tasks are generated at runtime; the amount of tasks depends of the input value; and they represent an independent set of instructions that can be performed in parallel. To return `x + y`, firstly the parallel executions of `fib( n-1 )` and `fib( n-2 )` must synchronize, which represents the joining of tasks. In the right, there is an illustration of the parallel execution of the Fibonacci for $n = 4$: the generation of the tasks happens while $n > 2$; then, the tasks join merging their results.

In the following, we will present the issues to provide abstract MPI tasks.

6.1.1 Issues of Abstract MPI Tasks

As shown in Section 4.3, the dynamic process creation can be used to provide the unfolding of the parallelism at runtime, generating MPI tasks on demand. Thus, the abstract tasks of explicit task parallelism can be mapped to MPI ones. Remember that MPI tasks have their own address space and, in general, are implemented as OS processes. To avoid misunderstood with the standard MPI tasks and those aimed in explicit task parallelism, we name this latter as abstract MPI tasks.

```
 1.int main( int argc, char **argv){
 2.  << declarations and inicializations >>
 3.
 4.  if( n < 2 )
 5.    MPI_Send( &n, 1, MPI_INT, 0, 1, parent );
 6.  else{
 7.    sprintf( argv[0], "%d", n-1 );
 8.    MPI_Comm_spawn( "spawn_fib", argv, 1, info, myrank,
                       MPI_COMM_SELF, &child[0], err );
 9.    sprintf( argv[0], "%d", n-2 );
10.    MPI_Comm_spawn( "spawn_fib", argv, 1, info, myrank,
                       MPI_COMM_SELF, &child[1], err );
11.    MPI_Recv( &x, 1, MPI_INT, MPI_ANY_SOURCE, 1, child[0], &st );
12.    MPI_Recv( &y, 1, MPI_INT, MPI_ANY_SOURCE, 1, child[1], &st );
13.    fibn = x + y;
14.    MPI_Send( &fibn, 1, MPI_INT, 0, 1, parent );
15.  }
16.}
```

Figure 6.2: Fibonacci with dynamic process creation (`spawn_fib.c`): two new processes are spawned to compute `n-1` and `n-2` in parallel; dependencies are satisfied through message exchanges – the parent wait for children results blocking in `MPI_Recv` in `child[0]` and `child[1]` intercommunicators while children send their results to the parent through `MPI_Send` in `parent` intercommunicator.

Figure 6.2 presents Fibonacci abstract MPI tasks, in which `MPI_Comm_spawn` replaces the recursive calls of Figure 6.1 and the messages solves dependencies. The declaration and initialization of variables and the MPI environment (`MPI_Init`, `MPI_Comm_get_parent`,...) were omitted to keep the source code simpler. The parent spawns children (launching two `spawn_fib` processes - lines 8 and 10), and blocks to synchronize with them (`MPI_Recv` in `child[0]` - line 11 - and `child[1]` intercommunicators - line 12). When children finish their computations, they return the results through `MPI_Send` in `parent` intercommunicator - line 14. Remembering, in the parent side, intercommunicators are returned from `MPI_Comm_spawn` and, in the children side, they are got by `MPI_Comm_get_parent`. The replacement of the recursive calls by `MPI_Comm_spawn` is the simplest way to develop explicit task parallel applications. This approach is widely used in shared-memory context, for instance, in TBB and OpenMP (Section 3.2.2). However, it rises in further issues to distributed-memory environments.

One of them is that the cost to create MPI tasks at runtime is greater than to create abstract tasks in shared-memory context. Basically, shared-memory APIs assign many fine-grained tasks to each thread (N tasks per thread - $N : 1$) and the number of threads is decided at runtime as the parallel hardware. In other words, the abstract tasks in shared-memory context are seen as the execution of few instructions (functions, or block of instructions, or even user-defined classes – saw in Section 3.2.2). Furthermore, the creation of these tasks has low costs once the assignment and the access to the input data happen through shared data structures.

However, the source code of Figure 6.2 generates one new MPI process to compute each abstract task (1 task per process - $1 : 1$). Thus, it implies in several process creations during the application execution. Abstract MPI tasks have higher

costs than to assign the execution of few instructions to threads, either in the time of creations, or in memory space required. Furthermore, the input data must be transmitted among distributed-memory regions (through argument or messages). Programming techniques can help to reduce these overheads, but they will not be eliminated.

Thus, we can observe two main problems of the straightforward replacement of the recursive calls for `MPI_Comm_spawn`: *(i)* it brings the assignment of one task to each MPI process (1 : 1) requiring many heavy-processes creations; and *(ii)* fine-grained tasks, as those in Figure 6.2, tend to be inefficient in MPI programs, since the cost to create a new process and transfer the input data is not outperformed by the computational time. Thus, the development of explicit task parallelism in MPI applications requires ways to reduce the overhead of multiple dynamic creations of MPI process. The next section explain how it can be solved adjusting the MPI tasks granularity.

6.1.2 Granularity of the Abstract MPI Tasks

Granularity is a qualitative measure of the ratio of the time spent by parallel programs computing to communicating (FOSTER, 1995). Thus, the efficiency of the parallel programs depends of the grain size, since it allows maximizing processor utilization and minimizing communication costs (FOSTER, 1995). In the context of MPI, static programs (those have an invariable number of processes defined at starting time – MPI-1 standard) the programmers control the granularity deciding the amount of work each process (*i.e.* each MPI task) must perform. The MPI-2 standard allows to create new MPI tasks at runtime, however the granularity definition kept as a programmers duty: dynamic MPI tasks are standard MPI processes spawned on demand. Thus, the same constraints must be supplied: the grain size of the MPI tasks must be large enough to outperform the communication times.

In the context of the explicit task parallelism, the goal is to focus the program development on the algorithmic issues, in such a way that programmers only identify the potential parallelism of their programs, while the runtime environment takes care of unfold it during the application execution according to the target architecture and input data. Furthermore, there are no information about the optimal granularity of these applications until runtime. The scheduling decisions are based on the availability of the processing elements (processors, cores), and on dependencies among tasks, both discovered at runtime. Adaptive approaches can benefit from dynamic mapping since they avoid machine specific parameters (CUNG et al., 2006).

Figure 6.3 shows a pseudo code of a generic abstract task according to the explicit task parallelism. The program splits the input in two halves and forks a new task for each half. A wait directive from the parallel API blocks the forker task until the execution of the forked ones finishes. Then, the results are merged and returned. We use `input.Size()` to return the input size - line 2, `input.Split()` to split the input in half - lines 4 and 5, `Fork(...)` to fork a task with some input - lines 4 and 5, `Wait_all_children_results()` (line 6) to block and wait for children, `output.Merge()` (line 7) to merge the children results, and `Sequential()` (line 9) to execute sequentially the workload.

Notice that the `THRESHOLD` guides the forking of new tasks (line 2). It can be used to indicate when the cost of new tasks creation is greater than the execution of their work sequentially. Considering the Fibonacci code in Figure 6.1: it is not

```
                /* Begin task */
 1. output_t Task ( input )
 2.   if input.Size() > THRESHOLD
 3.   then
 4.     output1 = Fork( Task( input.Split() ))
 5.     output2 = Fork( Task( input.Split() ))
 6.     Wait_all_children_results()
 7.     output.Merge( output1, output2 )
 8.   else
 9.     output = Sequential( input )
10.   end if
11.   return output
                  /* End task */
```

Figure 6.3: Pseudo code of a generic abstract task: tasks are dynamically forked until the recursive threshold is reached - each task receives half of the input data.

efficient to generate dynamic tasks for short values of n, since the computation is just a sum; otherwise, the program can stop the forking early and compute recursively (locally) the remaining work. For instance, to compute $n = 40$ is more efficient to generate new tasks until $n > 20$ and after to compute recursively than to generate new tasks until the default $n > 2$. Furthermore, the parallel APIs for explicit task parallelism such as TBB or OpenMP-3 recommend to use a threshold, which means that they warn about the granularity.

Consequently, to use the explicit task paradigm in MPI programs, it is required to keep the efficiency of the MPI tasks, *i.e.* to balance the distribution of the abstract tasks into the MPI tasks (MPI processes or MPI units of execution). Most of the parallel APIs for explicit task parallelism provides transparently the assignment and balancing of the abstract tasks. In this sense, Section 6.1.2.1 describes some challenges and issues to provide this transparency to the MPI users. On the other hand, to control the MPI tasks granularity is considered an MPI-user duty in the MPI standard. Thus, Section 6.1.2.2 shows how the users can implement such a control in their applications.

6.1.2.1 Adjusting the Grain Size of Abstract MPI Tasks: a Runtime-Level Approach

This section shows the requirements to provide the assignment and mapping of dynamically unfolded units of work, or abstract tasks, into MPI tasks, transparently to the users. In the previous sections, we defined abstract tasks and shown that the assignement of them is transparently provided in many explict tasks approaches. To achieve a similar behavior over the MPI context, such as the source code of Figure 6.2, the mean is to handle `MPI_Comm_spawn` calls.

Section 5.2.3 presented a scheduling library able to intercept `MPI_Comm_spawn` calls and query a scheduler daemon about the best physical location to the new processes. A similar mechanism can be used: to each `MPI_Comm_spawn`, the scheduler decides if spawns new MPI tasks (processes) or increases the granularity of the running ones, assigning further units of work. Figure 6.4 illustrates it: `MPI_Comm_spawn` is redefined to include a procedure to decide which actions must be done.

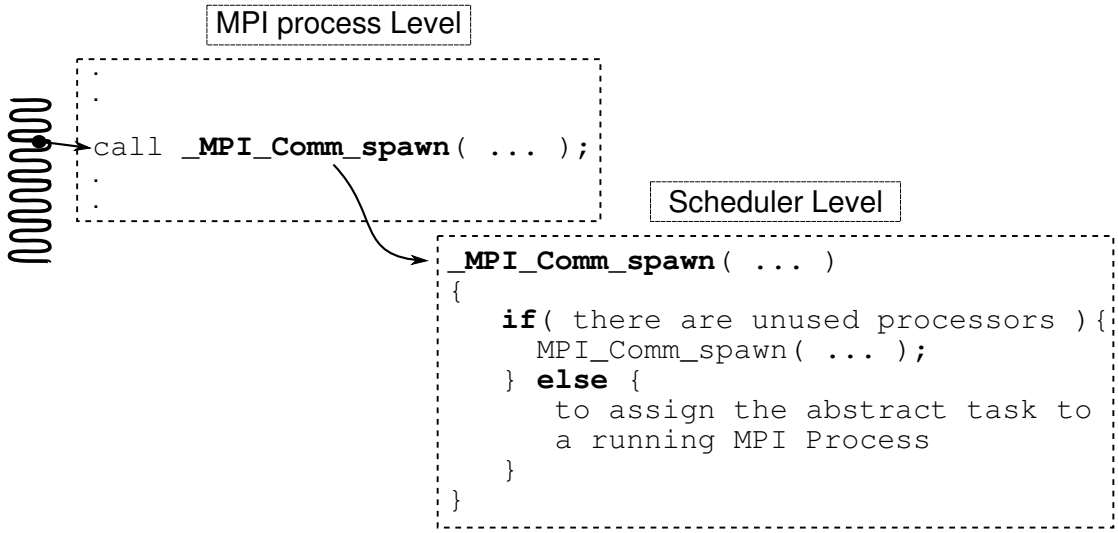


Figure 6.4: Scheduling abstract MPI tasks: *in MPI process level* – the redefinition of the `MPI_Comm_spawn` provided by a user-library allows to execute a procedure to deal with abstract task; *in the scheduling level* – it decides between to spawn a new MPI process or to assign work (abstract task) to running ones.

Looking at Figure 6.4, the first issue is the condition that guides the decision between the spawning or the assignment. In our illustration we test if there are unused processors, *i.e.* processors that do not run MPI processes. For efficiency reasons, the common goal is to destinate one MPI process to each processor. For instance, in OpenMPI this is the key condition to have an aggressive mode of execution, which ensures high performance. Furthermore, in nowadays multi-core architectures, each core is seen as a processor by the MPI distributions, then it is efficient to have as many processes as cores on the target architecture.

Another issue is to decide how to assign abstract tasks or the units of work into MPI processes, which requires some scheduling policy. For instance, an *Equipartition* policy (BUISSON et al., 2007) provides an egalitarian distribution of the abstract tasks among the running processes. *Adaptive* policies (GHOSE; KIM; KIM, 2005) are another option in which the decision is based on resource utilization estimates get through probing techniques. Thus, imbalances in task assignments can be solved at runtime. Additionally, in Section 6.3, we will discuss about the requirements to achieve load balancing in explicit task applications.

Furthermore, abstract tasks, or the units of work scheduled, in explicit task parallelism are function calls. Thus, the assignment of abstract tasks means to set the execution of a set of instructions for a specific input data to a target process. As each process gathers many abstract tasks, they can be sorted in a queue as the recursive calls constraints: lower levels calls are firstly executed to attend the upper levels dependencies. This queue can be used to implement Work Stealing strategy aiming to load balancing, as will show in Section 6.3.

```
                /* Begin task */
 1. output_t Task ( input )
 2.   level.Increase()
 3.   if input.Size() > THRESHOLD
 4.   then
 5.     if Nprocessors > level.Leaves()
 6.     then
 7.       output1 = Fork( Task( input.Split() )
 8.       output2 = Fork( Task( input.Split() )
 9.       Wait_all_children_results()
10.     else
11.       output1 = Task( input.Split() )
12.       output2 = Task( input.Split() )
13.     end if
14.     output.Merge( output1, output2 )
15.   else
16.     output = Sequential( input )
17.   end if
18.   return output
                  /* End task */
```

Figure 6.5: Pseudo code illustrating the adaptive task creation (*Adaptive*): dynamic MPI tasks are created while the number of leaves is less than the number of processors, thereafter tasks are recursively computed until a given threshold from that the computation is sequential.

6.1.2.2 Adjusting the Grain Size of Abstract MPI Tasks: an User-Level Approach

This section shows the development MPI applications following the explicit task parallelism. Here, programmers are responsible to adjust the abstract MPI task granularity. In other words, programmers implement a strategy in the application source code able to schedule the units of work into the available MPI tasks (MPI processes) at runtime.

The key to provide an user control of the abstract tasks assignment is the threshold. Previously, we show its use as a parameter of the parallelism extraction: new abstract tasks are generated just until their workload is large enough to bypass their creation costs. This naturally increases the granularity of the tasks. In this section we show that an additional stop condition (threshold) can guide the unfolding of the parallelism as the number of available processing elements (processors, cores). We call this approach as *Adaptive*.

Our approach distinguishes tasks according to their position in the call tree of D&C applications:

- **Leaves:** which are able to run immediately (dark gray circles in Figure 6.6);
- **Parents:** which keep blocked until the return of the children results (light gray circles in Figure 6.6).

Basically, the *Adaptive* approach spawns new MPI tasks (processes) until the number of leaves is equal to the number of available processing elements. Thus, it can provide a full utilization of the available processors. The additional thresh-

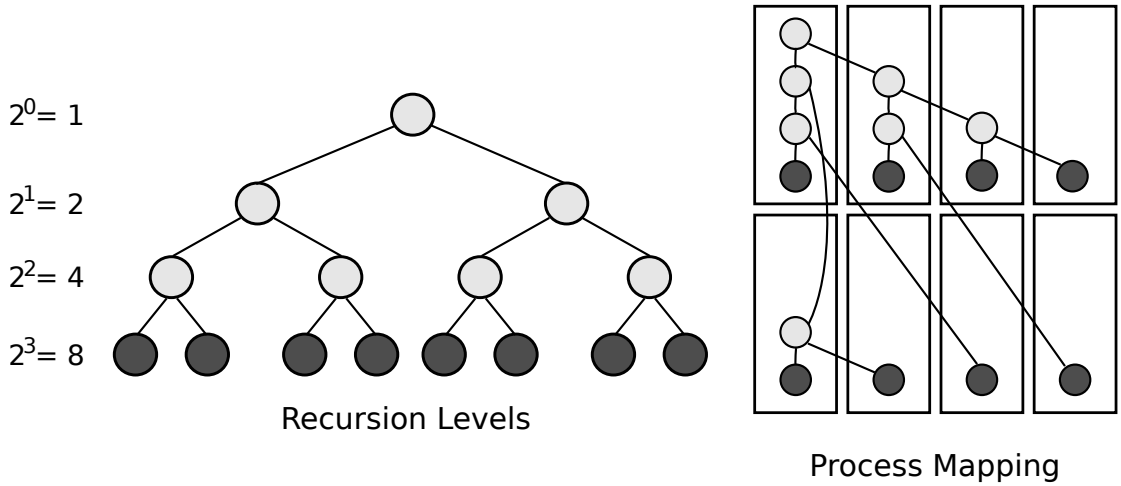


Figure 6.6: Controlling of the recursion levels and process mapping: *Adaptive* unfolds the parallelism until achieve 8 leaves (dark gray circle), which charge the available processors (rectangles).

old is responsible to control the spawning, which determines the exact level of the parallelism that spawns must stop without performance loss due to extra process creations. Thereafter, tasks perform recursive calls until achieve the algorithmic threshold, and finally, they compute the instructions sequentially. The execution of recursive calls increases the MPI process granularity and induces an egalitarian assignment of the workload: the input is split in two halves. However, *Adaptive* does not have any trivial implementation and demands changes in application source code.

Figure 6.5 shows a generic representation of the *Adaptive* approach. Notice that the `Fork(...)` (lines 7 and 8) means the calling of `MPI_Comm_spawn` in MPI applications. The additional stop condition (line 5) compares the number of processors (`Nprocessors`) with the number of leaves in the current recursion level (`level.Leaves()`). The first parameter of the condition is fixed and provided by the runtime environment. To second one, we add a data structure to represent the level of the recursive tree. At runtime, to each recursion level, a counter is incremented (`level.Increase()` - line 2). Thus, the number of running processes as leaves can be achieved (`level.Leaves()` of line 5) since it is 2^{level} where *level* is the current recursion level. Then, if the number of leaves is smaller than the number of processors, further processes are dynamically generated. Otherwise, tasks are executed recursively.

To exemplify the *Adaptive* approach, we illustrated the recursion levels of an explicit task application in Figure 6.6. There are 8 processors (rectangles) to execute the application. Thus, the *Adaptive* approach explore the application parallelism until reach the third recursion level, which provides $2^3 = 8$ leaves (dark gray circles). The remaining processes (7 parents) are hierarchically distributed on the tree (light gray circles). Thereafter, each left performs recursive calls as the algorithmic requirements. In the right, there is a representation of the process mapping of the *Adaptive* approach, in which each left is placed into an available processor.

6.1.2.3 Further Issues of Granularity

Regarding the previous approaches, they are focused on the adjustment of the MPI tasks granularity by the management of the recursive calls. Although they can improve the performance of MPI applications following the explicit task parallelism, there are other alternatives. A promising one is related to the current programming tendency aiming at multi-core architectures: the development of multi-threading programs. Moreover, today clusters of computers are composed by multi-core nodes, thus, parallel programs tends to be hybrid: intra-nodes computations are performed by threads with fast local communications; and inter-nodes exchanges happen through a fast standard inter-node interface (*e.g.*, MPI).

Furthermore, many parallel APIs for explicit task parallelism achieve their efficiency taking advantage of the shared-memory features. Thus, as the parallelism is extracted, threads can get units of work with a low overhead (accessing shared data structures). However, to have explicit task applications running efficiently on cluster environments, the inter-node communication, as well as its overhead, must be considered. Additionally, a common practice in HPC programming is to combine OpenMP threads (to achieve intra-node efficiency) with MPI (aiming at high performance inter-nodes interactions) (LEOPOLD; SÜSS; BREITBART, 2006; KRAWEZIK; CAPPELLO, 2006).

MPI-2 specifies the interaction between MPI functions and user-created threads. A fully thread-compliant MPI distribution supports `MPI_THREAD_MULTIPLE` level of thread safety, where multiple threads may make MPI calls at any time. Thus, an MPI program can combine processes and threads, both representing the MPI tasks. Although the performance gains using threads are obvious, the design of such MPI programs is not trivial because it must consider processes and threads mapping, and further, their safe communication. LIMA; MAILLARD (2009) investigate these issues to POSIX threads, providing a granularity control that decides between create threads or dynamic processes to avoid local nodes overloading. Furthermore, they manage communications among threads: `MPI_Send`/`MPI_Recv` communications converted to shared-memory accesses.

In this work, we will keep focused on controlling the granularity of explicit task MPI applications without take into account multi-threading issues. But these aspects are part of our future work perspectives.

6.2 Dependencies and Data Transfers among Abstract MPI Tasks

In standard MPI tasks, dependencies are solved through message exchanging taking advantage of its high performance communication interface. As could be seen in Section 4.3.2 the abstract MPI tasks adopt the same solution.

MPI dependencies are expressed as messages or collective barriers within a group of processes. `MPI_Comm_spawn` returns a communication handler (*intercommunicator*), which establishes an interconnection channel between two groups of processes: parent and children ones. This notion of intercommunicator is useful since the relation among processes is specific and simple. However, there is no trivial solution for global synchronizations because of the hierarchical dependency between children and parents (more details in Section 6.3).

```
 1. int main( int argc, char **argv ){
 2.   int n, Nproc, level;
 3.   MPI_Comm parent;
 4.
 5.   MPI_Init(&argc, &argv);
 6.   MPI_Comm_get_parent( &parent );
 7.   n = atoi( argv[1] );
 8.   Nproc = atoi( argv[2] );
 9.   level = atoi( argv[3] );
10.   level++;
11.   if( Nproc > pow( 2, level ) )
12.     spawn_fib( n, level, parent );
13.   else {
14.     int fibn;
15.     fibn = fib( n );
16.     MPI_Send( &fibn, 1, MPI_INT, 0, 1, parent );
17.   }
18.   MPI_Finalize();
19. }
```

Figure 6.7: *Adaptive* Fibonacci implementation (`adaptive_fib.c`): new Fibonacci processes are spawned until that there are as many MPI processes as processors. Thereafter, computations are recursively performed.

In the following, some aspects of the hierarchical communication required by MPI applications in the explicit task parallelism context are described in Section 6.2.1. Additionally, some opportunities to data transfers optimizations are discussed in Section 6.2.2.

6.2.1 Synchronizations by Blocking Communication

To understand the communication among MPI processes following the explicit task parallelism, Figure 6.7 applies the *Adaptive* approach in Fibonacci problem. Notice that there is another MPI process that spawns the first `adaptive_fib`, which was omitted. The execution of the `adaptive_fib` process starts setting up the MPI environment (`MPI_Init`- line 5) and getting the parent intercomunicator (line 6). Thereafter it stores the arguments: `n` is the Fibonacci number to be searched (line 7); `Nproc` is the number of processors (line 8); and `level` is the recursion level counter (line 9). The process increases the level counter (which has -1 at starting time) because it is a child - line 10.

Then, `adaptive_fib` tests the stop condition to verify the number of MPI processes and the number of processors - line 11. When there are fewer processes than processors, it calls `spawn_fib` (line 12) which is similar to Figure 6.2. Basically, this is a function that spawns two new `adaptive_fib` processes to compute `n-1` and `n-2` and blocks waiting for children results. Each new `adaptive_fib` process repeats the same instructions that its parent. Thus, it composes a hierarchical structure of blocked parents. When all processors are used, `adaptive_fib` processes start to compute recursively (`fib` - lines 13 to 17) as shown in Figure 6.1. When the results are ready, children send they back to their parents (line 16), unblocking them.

Figure 6.2 showed the use of `MPI_Recv` to block parent execution until children results are available. This point-to-point communication implements the parent and children synchronization, allowing that the parent continues to execute only after the dependencies with children are solved. In shared-memory APIs for explicit tasks parallelism, the synchronizations ensure that data is already updated and can be accessed by the parent. As in MPI context the environment is distributed-memory, a message exchange is required to allow the parent knows the children results. Thus, to implement synchronizations with a blocking point-to-point communications in MPI applications is a suitable alternative.

But, MPI offers other means to block the parent, which are more efficient than those showed in Figure 6.2. Instead of blocking in one `MPI_Recv` and after in another, forcing an receiving order of children results, the parent can start asynchronous receives. They are implemented through `MPI_Irecv` to start the asynchronous receiving; and `MPI_Waitany` to block waiting for the results. These MPI primitives were introduced in Chapter 4.

Depending on the features of problems being solved, advanced communication mechanism can improve the MPI applications performance. In the following we discuss about some possibilities.

6.2.2 Data Transfers Optimizations

MPI has powerful mechanisms to improve the parallel programs communication. One example is the collective communications, which can also be used to synchronize parent and children in explicit task applications. They provide a simpler and easier source code organization than to implement several point-to-point communications. Furthermore, explicit task applications can reduce communication costs using either *ad-hoc* strategies or specific platform optimizations of an MPI distribution.

An *ad-hoc* design uses application specific characteristics to increase granularity or locality, then reducing or eliminating process communications. For instance, when the MPI process performs recursive calls in *Adaptive* approach, the process granularity is increased and some communications are eliminated. In other words, computations performed recursively do not require communication as when they are computed by separate processes. Another way to increase the granularity is through hybrid programs combining threads and MPI processes, such as proposed by LIMA; MAILLARD (2009). Computations of hybrid programs require communications but they have low costs since several of them happen in a shared-memory context.

On the other hand, MPI distributions can implement optimizations selecting a communication mechanisms suited to the target architecture. For instance, to use sockets on clusters, and shared-memory copies on multi-core processors (BALAJI et al., 2009). These optimizations are becomming common on MPI distribution and the MPI applications desired by us can also take advantage of them.

6.3 On-line Scheduling of Abstract MPI Tasks

The explicit task parallelism is focused on the algorithm – programmers define which are the abstract tasks and their dependencies. However, it requires on-line scheduling once tasks are generated on demand and at runtime. Scheduling allows to fit abstract tasks into the available units of execution (processes or threads). Remembering that the amount of units of execution and abstract tasks are only know

at runtime, because the first depends on the architecture (shared or distributed-memory, how many processing elements,...) and the second depends on the input data. In the most popular programming tools that support the explicit task parallelism, this on-line scheduling is part of their runtime support.

In this section, we describe the main issues involving the scheduling of the explict task MPI applications. Section 6.3.1 focuses on the mapping of the abstract MPI tasks according to LAM/MPI and OpenMPI distributions. In addition, we present the challenges to provide load balancing to MPI applications with an explicit task behavior, as well as the constraints to implement Work Stealing to these applications.

6.3.1 Mapping of Abstract MPI Tasks

Remembering that abstract MPI tasks are MPI processes, which compute abstract tasks of the explic task MPI applications. Since these abstract MPI tasks are generated at runtime, their mapping is a key issue to ensure a good utilization of the available architecture. As shown in Section 6.1.1, we take advantage of the dynamic process creation to implement the unfolding of the parallelism at runtime. Thus, the mapping is dependent of the support provided by the MPI distributions.

In Section 5.2.2 we described the dynamic process mapping in two study cases of MPI distributions: LAM/MPI and OpenMPI. The same constraints are valid to explicit task parallelism in MPI applications.

- **LAM/MPI:** provides a Round Robin policy, however it performs poorly when multiples processes are spawned at the same time. In Section 5.2.3 we proposed a library to solve this problem and to provide a policy based on processes workload. Under this context, to execute MPI applications with an explicit task behavior on LAM/MPI, our proposed library must ensure a good cluster utilization. Furthermore, the workload-based approach may bring satisfactory results in this kind of MPI applications, once the underloaded processors receive the dynamic processes. Notice that to use a scheduling library to map dynamic processes and to implement a runtime-level adjusting of the granularity, as proposed in Section 6.1.2.1, are co-related solutions;
- **OpenMPI:** also provides a Round Robin policy, which performs as expected to multiple process spawning. Thus, the explicit task parallel applications can take advantage of the default mapping provided by OpenMPI. Although Round Robin does not offer ways to adapt the mapping to processing elements utilization, it can be efficient since some load balancing strategy is adopted. For instance, the user-level approach to adjust the process granularity as showed in Section 6.1.2.2.

Notice that our observations above focus on to define MPI process granularity, *i.e.* to decide how many abstract tasks each process must perform. However, in some cases, this is not enough to ensure performance because of the irregular nature of the problems, in which the execution time of the abstract tasks varies from one to another. Theses cases require an on-line scheduling strategy able to provide load balancing at runtime. The next section discusses about Work Stealing in MPI applications.

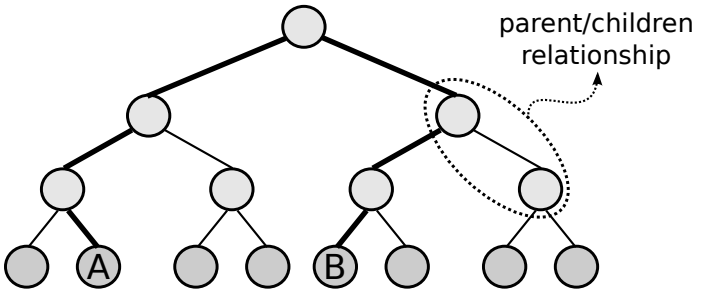


Figure 6.8: Communication hierarchy and the Work Stealing: parent/children communication relationship and the transmission of the stealing requests.

6.3.2 On-line Load Balancing for Abstract MPI Tasks

Section 3.3.2 showed the Work Stealing as the most adopted strategy to provide on-line scheduling thanks to its proved efficiency. So, many systems implemented this strategy on different contexts. Our previous work (PEZZI et al., 2007) implemented the Work Stealing inside of an MPI application source code, which spawn processes at runtime (Section 4.3.2). This work highlighted the constraints to implement the Work Stealing under this context, which are also valid to explicit task MPI applications as described in the following.

The first constraint is related to the communication relationship. As previously described, dynamic process creation brings hierarchical communications: processes have parent/children relationship (Figure 6.8). `MPI_Intercomm_merge` allows merging local and remote groups of processes in one intracommunicator. Although this primitive can be used to create a unique communication universe, it is a collective operation that requires global synchronizations in each process spawning. Thus, this alternative is impracticable for performance reasons.

The hierarchical communication structure impacts on transmission of stealing requests. As the victim is chosen randomly, many point-to-point communications may be required to transmit the requesting. For instance, suppose that *process A* chooses *process B* to steal tasks, as illustrated in Figure 6.8. To find it, the stealing request must go upward until the root, and after downward until the leaf, which means six point-to-point communications (dark branches). The response goes to the inverse path. In the worst case, *process B* has no tasks, and another victim must be chosen, requiring more communications. All these communications add extra costs, which become inviable the Work Stealing implementation under this context, because gains, in most cases, will be not outperformed by costs.

On distributed-memory environment, a well-known way to improve the Work Stealing performance is to consider the locality (NIEUWPOORT et al., 2006; GAUTIER; BESSERON; PIGEON, 2007). In MPI applications with dynamic processes, this is still more relevant due to the communication hierarchy. Thus, PEZZI et al. (2007) proposed a Hierarchical Work Stealing algorithm that guides the victim choice as the communication hierarchy. In few words, the stealer requests tasks from its parent: if the parent has tasks, it sends back and the stealing finishes; otherwise, the parent sends the request to the upper level, and this action is repeated until reach tasks or identify that there are no tasks to be compute. The stealing is propagated in the hierarchy only when the neighboring processes do not have tasks. Thus, the stealing may happen with less communications than choosing the victim randomly.

Besides the impact of multiple point-to-point communications in Work Stealing, naturally the cost to steal tasks in distributed-memory environments is greater than in shared-memory ones. Notice that in shared-memory, the stealing costs shared data structures accesses while in distributed, it costs inter-processes communications. Furthermore, the efficiency of the Work Stealing depends on the number of stealing operations, which must be lower than the number of operations in the critical path of the tasks graphs (BLUMOFE; LEISERSON, 1998). The higher is the application degree of parallelism, smaller is the critical path. Thus, even for highly parallel applications that require few stealing operations, their costs in MPI are greater than in shared-memory context, which impacts in the global parallel execution time. In this sense, the strategy must take into account the cost of stealing as a parameter to decide to steal or not.

Notice that the previous paragraphs are focus on Work Stealing to balance the workload distribution among MPI processes at runtime. Another work in our research group studied the usage of this strategy to map dynamic processes (MOR; MAILLARD, 2009). This work can be seen as an extension of the scheduling library (saw in Section 5.2.3), which provides a distributed scheduler for D&C MPI applications. It modified the dynamic process creation of MPICH2 distribution[1] in order to allow distributed decisions about physical locations of new processes. This implementation takes into account the MPICH2 features (such as its ring-based communication) and allows increasing the size of the problems that could be solved.

To provide a generic implementation of the Work Stealing to MPI applications with dynamic process creation is a challenge as can be seen by the constraints described in this section. Our research group has been studying this topic, which has a great relation to the applications aimed in this thesis. Thus, we decide to discuss the main conclusions about this adaptation of the Work Stealing, in which we were involved. However, its generic implementation was left to future researches due to its technical challenges.

Without an implementation of the Work Stealing, we are not able to deal with explicit task parallelism in MPI applications that have irregular workloads. However, we decide to test our approach that is able to define abstract MPI tasks, to control the granularity inside of the application source code, and satisfies dependencies through message exchanges. In this context, we implement some benchmarks, which are adopted for explicit task parallel APIs in shared-memory, to analyse our insights, as show in the next section.

6.4 Experimental Results

In this section, we will present our experimental results with MPI applications following the explicit task parallelism. Due to implementation issues, we are not able to provide a generic solution to unfold the parallelism of the MPI applications at runtime. But, we implemented solutions to these issues within applications source code. The goals of the results exposed in this section are:

- To validate the extraction of the parallelism of our MPI approach through a comparison with well-known shared-memory parallel API (Section 6.4.2);

[1]MPICH2 - *http://www.mcs.anl.gov/research/projects/mpich2/* last access in January 2011

- To analyse the performance of MPI applications following the explicit task parallelism in a distributed-memory environment (Section 6.4.3);
- To verify the impact of our *Adaptive* approach when compared to other other strategies to control the MPI process granularity (Section 6.4.4).

The results achieved shown that the explicit task parallelism paradigm can be adopted in MPI applications (running on distributed memory environments).

6.4.1 The Test Environment

All the results have been obtained on the French Grid'5000, tested from the site Porto Alegre, Brazil. Each node has two Intel© Xeon E5310 Quad Core 1.60 GHz processor (eight cores per node) and 16 GB of memory. We have used GCC version 4.3 with the OpenMPI distribution. All measures are the speedup relative to the best sequential running time, and each time is the mean of 30 executions. The standard deviation of the samples has always been smaller than 3%.

We have developed three test applications using the current MPI features in order to verify the efficiency of the MPI applications according to the explicit task parallelism. They are based on recursive algorithms whose each recursive call is a potential MPI task implemented as a dynamic process. The default OpenMPI mapping decides the physical location of spawned processes. In all applications, a recursive threshold is used to determine when sequential executions must replace the dynamic process creation. To determine the threshold to each application we estimate experimentally when the cost to create new processes is greater than the time to compute the workload sequentially.

The Fibonacci calculation is one of our test applications. Although its overall computation needs only few computational cycles, we aim to verify the behavior of our approach with a larger number of lightweight tasks. We are aware the Fibonacci is too simple to show an accurate measure of overall performance. However, its numerous lightweight tasks are very suitable to show how tasks behave when concerning granularity control and overall communication.

The second benchmark is the recursive implementation of the traditional matrix multiplication algorithm as described in Section 4.4.2. The input matrices, $A_{n\times n}$ and $B_{n\times n}$, are partitioned in two halves, generating four new abstract tasks per level. Each dynamic task computes one quadrant of the resulting matrix $C_{n\times n}$. When the results of the four abstract tasks are done, they are merged and sent to the upper level. This benchmark has heavier tasks that perform many computations and use memory intensively.

Merge sort is a well-known recursive sorting algorithm. Our parallel implementation works as the sequential version, where recursive calls are replaced by the forking of the abstract tasks. It divides an initial input of size n in two smaller parts of sizes $n/2$ for each abstract task. Then, this procedure is repeated until a threshold, from which the sequential algorithm is used. The conquer phase merges the results of two children and returns it to the upper level. Although most parallel merge sort implementations use shared-memory arrays for input and output numbers, the MPI version uses message passing for each new task. Experiments used n random numbers as input.

Table 6.1: Speedups of Fibonacci and Matrix Multiplication with MPI and OpenMP upon 1, 2, 4, and 8 cores of a multi-core machine.

Applications	**API**	**Number of Cores**			
		1	**2**	**4**	**8**
Fibonacci	MPI	1	1	2.62	3.40
	OpenMP	0.55	1.09	2.07	3.89
Matrix Multiplication	MPI	0.76	1.48	1.55	3.94
	OpenMP	1.17	1.77	2.25	2.47

6.4.2 Unfolding Parallelism of the MPI Applications

In this section, we validate the mechanism used to unfold the parallelism of MPI applications at runtime. The parallelism extracted should enable a good (linear) speedup. Therefore, we compare quantitatively the speedup achieved by our MPI applications with the speedup of a well-known API for explicit task parallelism: OpenMP. Notice that OpenMP generates dynamically as many fine-grained tasks as possible, which are scheduled on the set of threads by its runtime, ensuring the application performance. Aiming to achieve an optimum fit between the number of MPI processes and the available processing elements, MPI versions implement the *Adaptive* approach as described in Section 6.1.2.2.

The benchmarks used in this section are the Fibonacci computation and the matrix multiplication. The results represent the parallel computation of the $53th$ element in the Fibonacci sequence with a threshold equal to $46th$ element, which means that from that value computations are locally performed. Matrix multiplication results are the computation of two input matrices of 8192×8192 elements with a threshold of 256×256 elements. Besides, sequential executions took almost 1,240 seconds for Fibonacci and 80 seconds for matrix multiplication. Table 6.1 shows the speedups of the benchmarks using MPI and OpenMP upon a shared-memory processor with 1, 2, 4, and 8 cores. In our experiments, we used the `CPUSETs` handling to isolate set of cores in the multi-core machines.

Observing the speedups of Table 6.1, it can be verified that the greatest speedup is achieved by the MPI implementation of the Matrix Multiplication – near to **4** times better than the sequential version when are used 8 cores. Furthermore, for the same configuration, the MPI speedup is about **38%** better than the OpenMP one (3.94 for MPI and 2.47 for OpenMP). We believe that this speedups differences may be caused by cache issues, which explain the opposite behavior to Fibonacci. However, it is required a deepest investigation, that was left for future works.

Notice that the MPI versions have dynamic process creation and communications overheads without profit of the shared-memory advantages as OpenMP does. Thus, MPI versions achieved mostly lower speedups than OpenMP. However, the difference is low and MPI speedups are close to OpenMP (8% until 30% less performance), even without an efficient use of shared-memory features. In this sense, we can conclude that explicit task parallelism can be implemented on MPI applications achieving a performance coherent with well-known APIs such as OpenMP.

6.4.3 Performance of Explicit Task Parallelism in MPI Applications

This section aims to verify the behavior of MPI applications implementing the explicit task parallelism in distributed-memory environments. In this sense, we analyse the speedups of Fibonacci and Matrix Multiplication benchmarks in cluster architecture. We test two versions of each benchmark: one using our *Adaptive* approach; and another using a Naive implementation of task parallelism. Naive versions have only the threshold controlling the granularity of the tasks without take into account the amount of processing elements. This was used as an example in Section 6.1.1.

The speedups represent Fibonacci executions with the same input as in the previous section: $53th$ element in the sequence with a threshold of 46. Complying the requirements of distributed-memory executions, we increased the Matrix multiplication inputs to two matrices with 16384×16384 elements, keeping the threshold 256×256. The sequential execution time to Fibonacci keeps 80 seconds and almost 412 seconds to Matrix Multiplication.

Figures 6.9 and 6.10 show the speedups of Fibonacci and Matrix Multiplication, respectively, from 1 processor, with 8 cores, to 8 processors, resulting in 64 cores. To Fibonacci, the *Adaptive* approach reaches speedups mostly lower than the naive version (Figure 6.9). Adaptive speedup is greater than naive one only to 40, 48, and 56 cores and the best improvement is about 15%. This is because Fibonacci tasks involve few computations, and in this context to use a threshold is already enough to ensure good performance.

On the other hand, an adaptive granularity control in Matrix Multiplication allows a considerable gain of performance (Figure 6.10) – an improvement up to **90%** for 64 cores. As tasks take a long time computing, our simple effort to balance the load already impacts. This behavior expresses that scheduling issues are fundamental to achieve efficiency executing these MPI applications on distributed-memory environments.

6.4.4 Controlling the Granularity of the Abstract MPI Tasks

This section aims to analyse different mechanisms to provide a runtime adjustment of the MPI process granularity. In this sense, we compare three approaches: *Fork*, *Lazy*, and *Adaptive*. The latter was explained and used in the previous sections. The *Fork* approach is the Naive version described above: it spawns one new process to each abstract task until a given threshold. In *Lazy*, a process (the parent) spawns new processes to compute the abstract tasks, but it keeps one of them to compute locally. In this way, the parent also computes while wait for children results. Parent and children synchronize through a message receiving. As well as the other approaches, *Lazy* fork tasks until a given threshold.

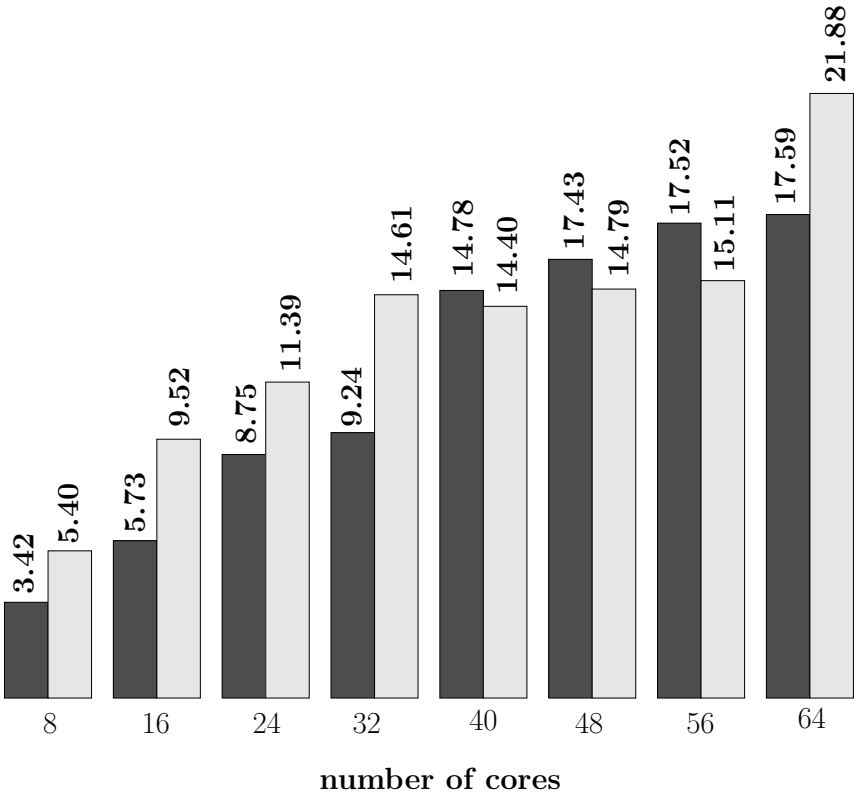


Figure 6.9: Speedup comparing the sequential execution time with naive and *Adaptive* MPI implementations of the Fibonacci calculation from 8 to 64 cores (1 to 8 nodes).

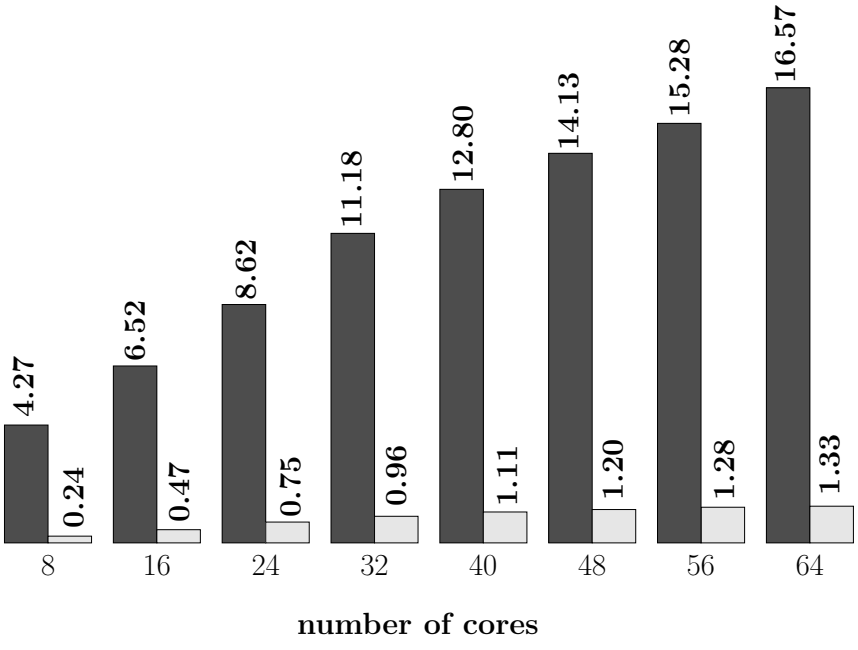


Figure 6.10: Speedup comparing the sequential execution time with naive and *Adaptive* MPI implementations of the Matrix Multiplication from 8 to 64 cores (1 to 8 nodes).

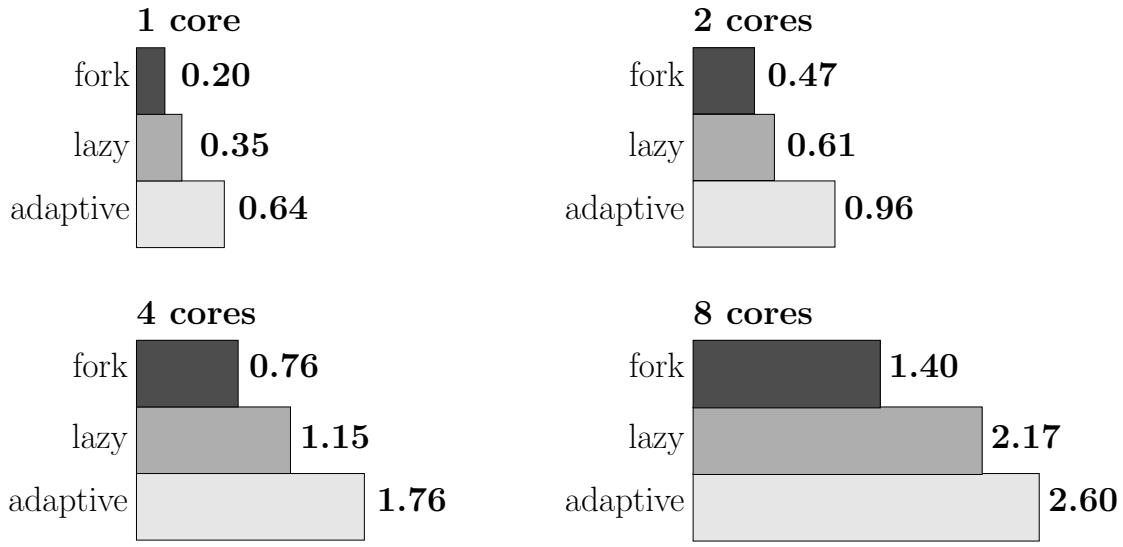


Figure 6.11: Merge Sort speedups of *Fork*, *Lazy*, and *Adaptive* approaches using an input of 400 million elements and a recursive threshold of 1 million elements, generating 1,023 tasks.

The Matrix multiplication benchmark keeps useful to measure *Lazy* and *Adaptive*'s efficiency. However, in this scenario, the didactic Fibonacci algorithm is useless, because benchmarking an I/O-bound parallel program while trying to speedup CPU-bound programs makes no sense; the gain obtained using tasks very short-lived is obvious. Therefore, we have to replace Fibonacci by a more suitable test-case when evaluating *Lazy* and *Adaptive* approaches: the Merge Sort.

The Matrix multiplication inputs are 8192×8192 elements and a threshold of 256×256 elements. The Merge Sort input is 400 million of random elements and a threshold of 1 million elements. Sequential executions took 80 seconds for Matrix Multiplication and 153 seconds for Merge Sort. Each graphic shown in this section represents the speedups of MPI applications designed with the three approaches (*Fork*, *Lazy*, and *Adaptive*). They are presented non-cumulatively, to better evaluate the individual gains. The execution is in shared-memory context with 1, 2, 4, and 8 cores, in which the isolation of cores happens through `CPUSETs` handling.

The *Adaptive* approach provides the greatest performance improvement as seen in Figures 6.11 and 6.12. This is because this approach impacts in two performance aspects: *(i)* it limits the number of process spawned, which limits also dynamic creation overhead into application performance; and *(ii)* performing recursion calls, synchronizations happen as functions return, instead of through message passing, which also take some overhead. The influence of these overheads can be seen in the difference from on approach to another for a given number of cores. *Fork* creates the greatest amount of processes and has the smallest speedups; *Lazy* spawns an intermediate amount of processes being the intermediate speedups; and *Adaptive* with less dynamic processes has the highest speedups.

Analysing the results, we can also observe that as the number of cores increases, the speedup of each approach also increases, thanks to the bigger degree of parallelism. For instance, to Merge Sort the *Fork* achieved 0.20 to 1 core, 0.47 to 2 cores, 0.76 to 4 cores, and 1.4 to 8 cores - *Lazy* and *Adaptive* follow the same tendency. However, comparing speedups of each approach for each number of cores, Merge

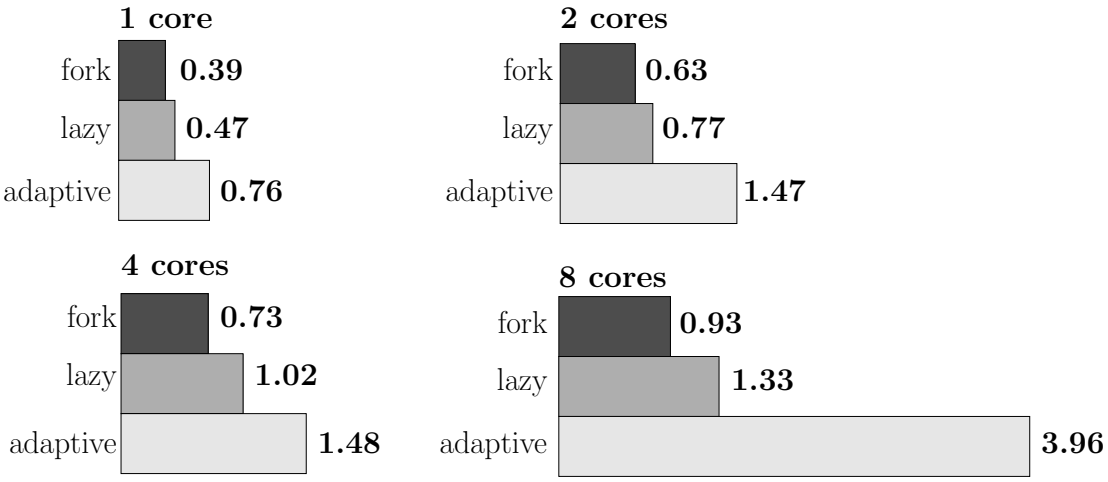


Figure 6.12: Matrix Multiplication speedups of *Fork*, *Lazy*, and *Adaptive* approaches using input matrices of 8192×8192 elements and a recursive threshold of 256×256 elements, generating 1,365 tasks.

Sort achieves the best gain when executed with 1 core: *Adaptive* is 68% more efficient than *Fork* (0.20 to 0.64). In other words, to reduce the number of running processes impacts significantly in this context. The smallest gain is executing over 8 cores: *Adaptive* is 46% better than *Fork*. This is because the increasing in the number of cores already provides relevant gains even to *Fork* approach (1 core: 0.20 to 8 cores: 1.4, *i.e.* the speedup with 8 cores is 85% better than with 1 core), thus, with 8 cores, the reduction of the number of processes can not improve significantly the speedup as with 1 core.

On the other hand, Matrix Multiplication has the best gain with 8 cores: *Adaptive* has a speedup 76% bigger than *Fork*. The smallest gain with 1 core: *Adaptive* is 48% better than *Fork*. As Matrix Multiplication is more CPU-intensive than Merge Sort, it is expected more gains over a larger number of processing elements, *i.e.* more computations performed by cycles. Notice that the gains when the degree of parallelism is increased to the *Fork* approach are smaller than to Merge Sort: speedup with 1 core: 0.39 and with 8 cores: 0.93, *i.e.* the speedup with 8 cores is 58% better than with 1 core. Then, to reduce the number of processes impacts more to the *Adaptive* approach than to *Fork* one: using 8 cores *Fork* has a speedup of 0.93 *vs.* 3.96 achieved by *Adaptive*.

However, to control the number of processes without provide load balancing is not efficient. This can be concluded locking at *Lazy* results: it limits the number of processes but its speedups are always nearer to *Fork*. This is because the processes workload is unbalanced (parent keeps computing large abstract tasks), which impact significantly in a CPU-intensive application.

Observing these results we can conclude that it is possible to implement task parallelism using MPI-2, since some control of MPI process granularity is offered. This goes against the claim that MPI is not suitable to task parallelism thanks to its task-spawning overhead. Furthermore, there is a technical constraint in the creation of large amounts of MPI tasks (processes): MPI distributions do not support more than few thousand processes running together in the same processor. This guided our choice about the number of dynamic MPI tasks and strengthens the need of a grain size control to implement explicit task parallelism in MPI.

Despite the efforts presented throughout this section, MPI does not address optimizations on task creation, since the standard aims to be a portable interface. It assumes an external solution from the MPI distribution or the programmer for scheduling and granularity control, which are partially addressed here. Almost all results confirmed the overhead on task creation and scheduling; our *Adaptive* approach probably offers performance gains to Fork-Join applications.

6.4.5 Blueprint for Explicit Task Parallelism in MPI

MPI has not been originally designed to support explicit task parallelism. However, the situation was the same for OpenMP, yet its forum has found means to define tasks and their dependecies in the native interface. We belive that the same result could be achieved to MPI, considering that a new standard, MPI-3, is being discussed now.

`MPI_Comm_spawn` is a natural and native way to define and fork new MPI tasks. The original MPI 1.2 norm already defined MPI tasks independently of the notion of process (in the OS sense). However, the current `MPI_Comm_spawn` is meant to run new images of MPI binaries, *i.e.* to create new processes. An interesting improvement would be to allow the call of functions instead of, or together with MPI binaries. Thus the programmer could easily control the granularity choosing between processes creation or recursive calls (as introduced in Section 6.1.2.2). Furthermore, `MPI_Comm_spawn` would be closer to the model of POSIX Threads facilitating the integration of both in MPI programs.

The choice between spawning functions or binaries could be let to the programmer by the use of a special field of `MPI_Info` parameter passed to `MPI_Comm_spawn`, or alternatively by providing a new `MPI_Datatype` that would describe the function, or binary, to be run. The user could also provide hints to set the granularity (threshold). This description would include the arguments of the task. Either they can be described, as in the current `MPI_Comm_spawn`, by the `argv/argc` arguments; or by the appropriate `MPI_Datatypes` (one per argument). With the latter solution, parent/children exchanges of the input and output would be trivial.

Probably, the solution based on the current `MPI_Comm_spawn` and an extended use of the `MPI_Info` parameter, is more coherent with MPI-2.

6.5 Conclusion

In this chapter, we presented the efforts to provide evolving MPI applications through the explicit task parallelism. These applications are able to unfold the parallelism at runtime adapting the application execution according to the architecture and input data. Our proposal takes advantage of the dynamic process creation to implement the on-the-fly unfolding of the parallelism. However the development of these applications requires 3 main aspects:

- **To define abstract MPI tasks:** Section 6.1 illustrated, with an example, the main issues to implement abstract MPI tasks as required by the explicit task parallelism paradigm. Furthermore, we shown that to adjust the MPI process granularity is the key to achieve performance running these application in shared-memory environments. We presented two approaches to control granularity of the abstract MPI tasks: *runtime-level* aiming to offer adjustments

Table 6.2: Achieved improvements comparing *Adaptive*, *Fork*, and *Lazy* approaches.

Application	**the best gain**	**the smallest gain**	***Fork* gain**
Merge Sort	68% with 1 core	46% with 8 cores	85%
Matrix Multiplication	76% with 8 cores	48% with 1 core	58%

transparently to the programmers; and a *user-level* in which programmers implement the adjustment of the granularity in the application source code, following some strategy. In addition, we discussed about the controlling of the granularity using threads, and thus, taking advantage of the multi-core architectures;

- **To solve dependencies among abstract MPI tasks:** Section 6.2 explained how to use the intercommunicator between parent and children processes and the communication primitives to solve dependencies in the target applications. Moreover, we discuss about the use of MPI high performance exchanges to improve the application performance;
- **To schedule abstract MPI tasks at runtime:** Section 6.3 shown the two main issues of abstract MPI tasks scheduling, both defined at runtime: their *mapping* on the available processing elements, which is related to the MPI distributions; and their *load balancing* to allows an on-the-fly adaptation to workload variations. We described the challenges to develop a generic implementation of the Work Stealing strategy.

Our experimental results presented in Section 6.4, validated our proposal of explicit task parallelism in MPI applications: to use dynamic process creation to generate abstract MPI tasks (MPI processes) on demand as the exploration of the parallelism. This proposal can not be implemented without: to provide synchronizations among abstract MPI tasks, which are implemented as messages exchanges; and to control the MPI process granularity. Thus, we take care of these issues within the applications source code. Although we do not have a generic solution to achieve explicit task parallelism in MPI applications, our experiments showed that this paradigm can also be efficient on distributed-memory environments, as in shared-memory ones.

We used 3 benchmarks in our experiments. We validate the mechanism used to extract the parallelism at runtime: to spawn processes as the input data and the available processing elements, which provides granularity adjustments on demand. In our experiments, the *Adaptive* approach was able to achieve a gain of **38%** when compared with the parallelism extraction provided by OpenMP (Section 6.4.2). Notice that this gain is achieved even without take advantage of the shared-memory properties as OpenMP does. Thus, the spawning of the MPI processes is able to provide the adaptive behavior expected from explicit task applications in the MPI context.

Furthermore, we also verify the behavior of the MPI applications following the explicit task parallelism in cluster environments, which achieved, in our tests, an improvement of **90%** to the CPU-intensive application when compared to a Naive implementation (Section 6.4.3). This result confirms that explicit task parallelism can be efficient even in distributed-memory environments. However, to control the amount of work that must be performed by each abstract MPI tasks is a key issue.

In this sense, we compared different approaches to control the granularity of the abstract MPI tasks: *Fork* (as many MPI processes as abstract tasks), *Lazy* (to spawn an MPI processes to an abstract task while compute the other locally), *Adaptive* (as many MPI processes as processing elements). Table 6.2 presents the gains achieved comparing the approaches on two different applications: the second column has the best gains of *Adaptive* compared to *Fork*; the third has the smallest gains; and the fourth has the improvement achieved by the *Fork* approach from 1 core until 8 ones (Section 6.4.4). We can conclude that to take into account the degree of parallelism of the target architecture could be more efficient to a CPU-intensive application such as Matrix Multiplication.

The main contribution of this chapter was to confirm our hypothesis that MPI applications can be developed according to the explicit task parallelism paradigm thanks to MPI-2 features. However, to provide the unfolding of the parallelism at runtime and transparently to the users is still an open issue to be researched. Furthermore, we identify opportunities to extend the MPI standard in such a way to include in its native interface ways to define abstract tasks and solve their dependencies.

7 CONCLUSION

As the current tendency on parallel programming, this thesis aimed to provide and support adaptive MPI applications. In this way, the range of problems addressed by the MPI standard can be increased. Our hypothesis is that the MPI-2 features can be used to provide adaptability, in special the dynamic process creation. We focused our study on two aspects:

- **To adapt the application execution to changes in availability of the processors.** Due to the increasing use of parallel architectures, users compete for resources. So, the number of available processors may vary according to users' demands. When the set of processors used by an application change, it must adapt itself to allow their efficient utilization, *i.e.* it must be *Malleable*. Thus, the application must expand (to use further processors) and shrink (to release some processors), without compromise its execution and the correctness of its results. In this sense, the dynamic process creation is used to implement the growth actions or to expand the number of used processors;
- **To adapt the unfolding of the parallelism of the applications to the available parallel architecture and input data.** With the popularity of the multi-core architectures, parallel applications must be able to adapt their execution to any amount of processing elements. Thus, the *Explicit Task Parallelism* comes as a powerful programming paradigm: it extracts the algorithm parallelism at runtime, according to the degree of parallelism in the target architecture and the input data. Dynamic process creation is used to implement the unfolding of the parallelism: new processes are spawned on demand and dependencies are solved through message exchanges.

Both aspects aimed in this thesis represent a new usage of the MPI standard. Thus, the first issue is how to develop these adaptive MPI applications. Chapter 4 explained the usage of `MPI_Comm_spawn` primitive that creates processes at runtime, and the resultant inter-processes communication relationship. It is presented an analysis of the requirements to support malleability in SPMD and Master/Worker program structures. Prototypes of the required procedures to support malleability are also presented. The basic ones are: `malleability_handle` to identify changes in the set of processors and launch the adaptive action; `growth_action` that spawns processes on further processors and give some workload to them; and `shrinkage_action` to release the required processors ensuring the application correctness (transferring dieing process workload).

The requirements to support explicit task parallelism in MPI applications are also described in Chapter 4. We focus on Fork/Join program structure and D&C algorithms, which are widely used in explicit task parallelism because each recursive call represents an abstract task. Our proposal is to replace recursive calls by dynamic process creations. Thus we map abstract tasks to MPI tasks. Synchronizations among abstract tasks (*i.e.* the function returns) are provided by message exchanges. Under this context, the key point to have efficient MPI applications following the explicit task parallelism is the scheduling: to *map* dynamic processes into the available processing elements; and to *adjust* the MPI process granularity to ensure load balancing.

Moreover, we illustrated how to use the `MPI_Comm_spawn` to support adaptability through practical examples of Matrix Multiplication implementations, including malleability and explicit task parallelism with the aimed program structures. Once the technical issues to develop adaptive MPI applications were described, a common point is the specific requirements to support each class of application: malleable ones require interaction with the RMS to know about processors availability; and explicit task ones require some mechanism to control the processes granularity.

Malleable applications adapt themselves to volatile processors, but the information about processors availability must comes from the runtime environment. As RMS manages the resources of clusters, it can also provide the information required by malleable applications. In Chapter 5, we showed the OAR resource manager, its functionality linked to the malleability support (*Best Effort* jobs), and its policy to manage volatile processors. *Best Effort* are low-priority jobs that can be finished at any time and without previous announcement. Thanks to this kind of job, we could implement malleable jobs in OAR: to launch new *Best Effort* jobs in further processors to increase the number of used processor; and to kill some *Best Effort* jobs on the required processors to decrease the number of processors.

We observed in our experiments using benchmarks that malleable jobs could improve up to **25%** of cluster utilization when compared to a non-malleable approach. This gain was measured on a real cluster: we used real workload traces to charge our cluster processors representing a production cluster usage and combined malleable and *Best Effort*-moldable jobs together with the rigid ones to measure their impact on cluster utilization. However, the execution of the malleable applications on clusters environments depends on interactions between RMS and applications to update the processors availability. OAR interacts with malleable MPI applications through our dynamic process scheduler, which is responsible to map the dynamic process and to launch the adaptive actions. Thanks to this interaction, we could test our malleable MPI applications in cluster production environments.

Evolving MPI applications aimed in this thesis spawn processes at runtime extracting the parallelism on demand. Chapter 6 defined the abstract MPI tasks, showing: the spawning of new tasks as given conditions; the blocking to synchronize parent and children; and the returning results to satisfy dependencies. In addition, it is shown the relevance of a granularity control because MPI tasks have creation and communications overheads that must be overlapped. Two approaches are shown: a runtime-level able to control the granularity without programmers' intervention; and user-level that programmers implement the adjustment of the granularity inside of application source code. Furthermore, these applications can take advantage of high performance communication mechanisms to synchronize and solve dependencies.

As the unfolding of parallelism happens at runtime, MPI applications following the explicit task parallelism require on-line scheduling to: map the new processes into the available processors; and balance the workload among processes. The first is addressed by the MPI distributions, and to the second we described the challenges and constraints to implement a Work Stealing strategy as these applications features. Although we are not able to implement a generic version of the Work Stealing to provide a transparent load balancing, we implement the adjustment of the workload distribution within the application source code.

Our non-generic approach allows to verify three main aspects: *(i)* we developed MPI applications able to unfold the parallelism at runtime, in which our experiments using benchmarks arrive to be **38%** more efficient than OpenMP on shared-memory context; *(ii)* on distributed-memory environments the adoption of the explicit task paradigm can improve CPU-intensive application performance – our experiments with benchmarks achieve improvements up to **90%** when compared to a Naive implementation; *(iii)* to control the spawning of the MPI processes, adjusting the MPI task granularity, to the specific degree of parallelism of the target architecture ensure gains to application performance – our experiments show from **46%** to **68%** for a memory-intensive application (Merge Sort), and from **48%** to **76%** for a CPU-intensive application (Matrix Multiplication). These results confirm our hypothesis: explicit task parallelism can be efficient even on distributed-memory environments. However, to offer it transparently to the users is still an open challenge.

7.1 Contributions

The main contributions of this thesis are:

- **The hypothesis that MPI-2 features can be used to implement adaptive MPI applications was confirmed.** Dynamic process creation adds the required flexibility to the MPI applications allowing performing adaptive actions at runtime;
- **This thesis proposed the design of malleable MPI applications.** We presented the procedures to implement adaptive actions taking advantage of the dynamic process creation, allowing the execution of MPI application using volatile processors. Furthermore, the specific issues of SPMD and Master/-Worker program structures were described;
- **To support malleable MPI applications, this thesis shows the integration of a generic RMS and our dynamic process scheduler.** This integration is the key to execute malleable applications in production cluster environments, because the adaptive actions depend on processors availability, which are managed by RMS systems. It was shown the required information exchanges and the ways to implement it;
- **This thesis proposed the design of explicit task parallelism in MPI applications.** To allow unfolding the application parallelism according to the architecture and input data properties, we defined the abstract MPI tasks, solved their dependencies through message exchanges, and implement an on-line approach to balance the workload. Our solution was designed inside of the applications source code, showing that explicit task parallelism is a promising programming paradigm to shared-memory environments.

7.2 Future Works and Perspectives

During the development of this thesis, we achieve many research and technical challenges. We focus on solve some of them to confirm our hypothesis, but there are others that we can not attended. Furthermore, the achieved results show also new research perspectives. In the following, we describe the most relevant ones.

- We developed malleable MPI applications using simple solutions to implement adaptive actions. Thus, a future work is to analyse the impact of advanced solutions such as to use a mechanism of checkpointing-restart in shrinkage actions to do not loss already computed data; and to use migration to transfer processes on adaptive actions instead of kill them;
- Parallel programming to multi-core architectures is naturally multi-threaded. We highlight the improvements that can be achieved with hybrid programs that combine MPI processes and threads (OpenMP or Posix ones). Our research group has being developing studies with this focus and that is a promising perspective of future works. In addiction, hybrid programs can also aim other kind of architectures such as GP-GPUs ones. Notice that to perform efficiently in those heterogeneous environments adaptability is the key issue;
- Our research group has being studying the adaptation of the Work Stealing strategy to features of the MPI applications that spawn processes at runtime. This study aims to propose a generic implementation of the strategy, but many constraints are being achieved. Thus, the research group continues to investigating these issues;
- Considering hybrid programs aiming to multi-core architectures, another research perspective is to narrow the Work Stealing target on the intra-node scope (shared-memory), where it is proven efficient. Thus, combining another strategy to distribute the workload on the inter-node scope (distributed-memory), such as our *Adaptive* approach, with the Work Stealing intra-node scheduling, it can be supported adaptive applications transparently on cluster environments;
- Related to the explicit task parallelism in MPI applications, our research is still in an initial phase. Thus, we intent to continue verifying its efficiency, including comparison with other APIs such as KAAPI, and extend our proposal that control the extraction of the parallelism at runtime-level;

REFERENCES

AYGUADÉ, E. et al. The Design of OpenMP Tasks. **IEEE Trans. Parallel Distrib. Syst.**, [S.l.], v.20, n.3, p.404–418, 2009.

BALAJI, P. et al. Toward message passing for a million processes: characterizing mpi on a massive scale blue gene/p. **Computer Science-Research and Development**, [S.l.], v.24, n.1, p.11–19, 2009.

BENDER, M. A.; RABIN, M. O. Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. In: TWELFTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES - SPAA, Bar Harbor, Maine, USA. **Anais...** [S.l.: s.n.], 2000. p.13–21.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. **Journal of Parallel and Distributed Computing**, [S.l.], v.37, n.1, p.55–69, 1996. (An early version appeared in the *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, 1995.).

BLUMOFE, R. D.; LEISERSON, C. E. Space-efficient scheduling of multithreaded computations. **SIAM Journal on Computing**, [S.l.], v.27, n.1, p.202–229, 1998.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling Multithreaded Computations by Work Stealing. **Journal of the ACM**, [S.l.], v.46, n.5, p.720–748, 1999.

BOERES, C. et al. Efficient hierarchical self-scheduling for MPI applications executing in computational Grids. In: MGC '05: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING, New York, NY, USA. **Anais...** ACM Press, 2005. p.1–6.

BOLZE, R. et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. **Int. Journal of High Performance Computing Applications**, [S.l.], v.20, n.4, p.481–494, 2006.

BUISSON, J. **Adaptation dynamique de programmes et composants parallèles**. 2006. Tese (Doutorado em Ciência da Computação) — INSA de Rennes.

BUISSON, J.; ANDRÉ, F.; PAZAT, J.-L. A Framework for Dynamic Adaptation of Parallel Components. In: INTERNATIONAL CONFERENCE OF PARALLEL COMPUTING: CURRENT & FUTURE ISSUES OF HIGH-END COMPUTING,

Department of Computer Architecture, University of Malaga, Spain. **Proceedings...** Central Institute for Applied Mathematics: Jülich: Germany, 2005. p.65–72. (John von Neumann Institute for Computing Series, v.33).

BUISSON, J.; ANDRÉ, F.; PAZAT, J.-L. Afpac: enforcing consistency during the adaptation of a parallel component. **Scalable Computing: Practice and Experience**, [S.l.], v.7, n.3, p.83–95, September 2006.

BUISSON, J.; ANDRÉ, F.; PAZAT, J.-L. Supporting adaptable applications in grid resource management systems. In: IEEE/ACM INTERNATIONAL CONFERENCE ON GRID COMPUTING (GRID 2007), 8., Austin, Texas, USA. **Proceedings...** IEEE, 2007. p.58–65. ISBN 1-4244-1560-8.

BUISSON, J. et al. Scheduling Malleable Applications in Multicluster Systems. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING. **Anais...** IEEE Computer Society, 2007. p.372–381.

CAPIT, N. et al. A batch scheduler with high level components. In: INT. SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 5., Cardiff, UK. **Anais...** IEEE, 2005. p.776–783.

CERA, M. C. et al. Improving the Dynamic Creation of Processes in MPI-2. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 13TH EUROPEAN PVM/MPI USER'S GROUP MEETING, Bonn, Germany. **Anais...** [S.l.: s.n.], 2006. p.247–255. (Lecture Notes in Computer Science, v.4192/2006). issn:0302-9743.

CERA, M. C. et al. Scheduling Dynamically Spawned Processes in MPI-2. In: JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 12TH INTERNATIONAL WORKSHOP, Saint Malo, France. **Anais...** Springer, 2006. p.33–46. (Lecture Notes in Computer Science, v.4376). 3-540-71034-5.

CERA, M. C. et al. Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING AND NETWORKING, 11., Kolkata, India. **Proceedings...** Springer, 2010. p.242–257. (LNCS, v.5935).

CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using OpenMP**: portable shared memory parallel programming. Cambridge, MA: MIT Press, 2008. (Scientific and Engineering Computation Series).

CUNG, V.-D. et al. Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving. In: TRANSGRESSIVE COMPUTING TC'2006, Granada, Spain. **Anais...** [S.l.: s.n.], 2006. p.131–148.

DESELL, T.; MAGHRAOUI, K. E.; VARELA, C. A. Malleable applications for scalable high performance computing. **Journal of Cluster Computing**, [S.l.], v.10, n.3, p.323–337, 2007.

DU, C. et al. A Runtime System for Autonomic Rescheduling of MPI Programs. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP 2004), 33., Montreal, Quebec, Canada. **Anais...** IEEE Computer Society, 2004. p.4–11.

DUTOT, P.-F. et al. Scheduling on large scale distributed platforms: from models to implementations. **International Journal of Foundations of Computer Science**, [S.l.], v.16, n.2, p.217–237, 2005.

DUTOT, P.-F.; MOUNIÉ, G.; TRYSTRAM, D. Scheduling Parallel Tasks: approximation algorithms. In: LEUNG, J. T. (Ed.). **Handbook of Scheduling**: algorithms, models, and performance analysis. USA: CRC Press, 2004. p.26–1–26–22.

FEITELSON, D. G.; RUDOLPH, L. Toward Convergence in Job Schedulers for Parallel Supercomputers. In: FEITELSON, D. G.; RUDOLPH, L. (Ed.). **Job Scheduling Strategies for Parallel Processing**. [S.l.]: Springer-Verlag, 1996. p.1–26. (Lecture Notes in Computer Science, v.1162).

FOSTER, I. **Designing and Building Parallel Programs**. [S.l.]: Addison-Wesley, 1995.

FOSTER, I. Globus toolkit version 4: software for service-oriented systems. **Journal of Computational Science and Technology**, [S.l.], v.21, n.4, p.523–530, 2006.

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, NY, USA. **Anais...** ACM, 2007. p.15–23.

GEORGIOU, Y. **Contributions for Resource and Job Management in High Performance computing**. 2010. Tese (Doutorado em Ciência da Computação) — Université Joseph Fourier.

GEORGIOU, Y.; RICHARD, O.; CAPIT, N. Evaluations of the Lightweight Grid CIGRI upon the Grid5000 Platform. In: THIRD IEEE INTERNATIONAL CONFERENCE ON E-SCIENCE AND GRID COMPUTING E-SCIENCE '07, Washington, DC, USA. **Anais...** IEEE Computer Society, 2007. p.279–286.

GHAFOOR, S. K. **Modeling of an adaptive parallel system with malleable applications in a distributed computing environment**. 2007. Tese (Doutorado em Ciência da Computação) — Mississippi State University, Mississippi State, MS, USA. Adviser-Banicescu, Ioana.

GHOSE, D.; KIM, H. J.; KIM, T. H. Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources. **IEEE Transactions on Parallel Distributed Systems**, Piscataway, NJ, USA, v.16, n.10, p.897–907, 2005.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: Portable Parallel Programming with the Message Passing Interface. Cambridge, Massachusetts, USA: MIT Press, 1994.

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2 Advanced Features of the Message-Passing Interface**. Cambridge, Massachusetts, USA: The MIT Press, 1999. 382p.

HAMILTON, J. D. **Time Series Analysis**. Princeton, NJ, USA: Princeton University Press, 1994.

HEYMANN, E. et al. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In: GRID'00: PROCEEDINGS OF THE FIRST IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, London, UK. **Anais...** Springer-Verlag, 2000. p.214–227.

HUANG, C. et al. Performance Evaluation of Adaptive MPI. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING 2006. **Proceedings...** [S.l.: s.n.], 2006.

HUANG, C.; LAWLOR, O. S.; KALÉ, L. V. Adaptive MPI. In: INTERNATIONAL WORKSHOP OF LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC 2003), 16., College Station, TX, USA. **Anais...** Springer, 2003. p.306–322. (Lecture Notes in Computer Science).

HUNGERSHFÖER, J.; STREIT, A.; WIERUM, J.-M. **Efficient Resource Management for Malleable Applications**. [S.l.]: Paderborn Center for Parallel Computing, 2001. (TR-003-01).

HUNGERSHÖFER, J. Increased Scheduling Quality by Utilizing the Flexibility of Malleable Jobs. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS, 17. **Anais...** [S.l.: s.n.], 2004. p.72–77.

JANSEN, K.; ZHANG, H. Scheduling malleable tasks with precedence constraints. In: SPAA'05: PROCEEDINGS OF THE 17TH ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, New York, NY, USA. **Anais...** ACM Press, 2005. p.86–95.

KALÉ, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on C++. In: OOPSLA '93: PROCEEDINGS OF THE EIGHTH ANNUAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, New York, NY, USA. **Anais...** ACM Press, 1993. p.91–108.

KALÉ, L. V.; KUMAR, S.; DESOUZA, J. **An Adaptive Job Scheduler for Timeshared Parallel Machines**. [S.l.]: Parallel Programming Laboratory, Dep. of Computer Science, University of Illinois at Urbana-Champaign, 2000. (00-02).

KRAWEZIK, G.; CAPPELLO, F. Performance comparison of MPI and OpenMP on shared memory multiprocessors. **Concurrency and Computation: Practice and Experience**, [S.l.], v.18, n.1, p.29–61, jan 2006.

LEISERSON, C. E. The Cilk++ concurrency platform. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 46. **Proceedings...** ACM, 2009. p.522–527.

LEISERSON, C. E.; MIRMAN, I. B. **How to Survive the Multicore Software Revolution**. [S.l.]: Cilk Arts, Inc, 2008. e-book.

LEOPOLD, C.; SÜSS, M. Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 13TH EUROPEAN PVM/MPI USER'S GROUP MEETING, Bonn, Germany. **Anais...** Springer, 2006. p.285–292. (LNCS, v.4192).

LEOPOLD, C.; SÜSS, M.; BREITBART, J. Programming for Malleability with Hybrid MPI-2 and OpenMP: experiences with a simulation program for global water prognosis. In: EUROPEAN CONFERENCE ON MODELLING AND SIMULATION, Bonn, Germany. **Anais...** [S.l.: s.n.], 2006. p.665 – 670.

LEPÈRE, R.; TRYSTRAM, D.; WOEGINGER, G. J. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. **Int. Journal of Foundations of Computer Science**, [S.l.], v.13, n.4, p.613–627, 2002.

LI, H.; GROEP, D. L.; WOLTER, L. Workload Characteristics of a Multi-cluster Supercomputer. In: JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 10. **Anais...** Springer, 2004. p.176–193. (LNCS).

LIMA, J. V. F.; MAILLARD, N. Online mapping of MPI-2 dynamic tasks to processes and threads. **International Journal of High Performance Systems Architecture**, [S.l.], v.2, n.2, p.81–89, 2009.

LITZKOW, M.; LIVNY, M.; MUTKA, M. Condor - A Hunter of Idle Workstations. In: INTERNATIONAL CONFERENCE OF DISTRIBUTED COMPUTING SYSTEMS, 8. **Proceedings...** [S.l.: s.n.], 1988.

MAGHRAOUI, K. E. et al. Dynamic Malleability in Iterative MPI Applications. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID 2007), 7. **Anais...** IEEE Computer Society, 2007. p.591–598.

MAGHRAOUI, K. E. et al. Malleable iterative MPI applications. **Concurrency and Computation: Practice and Experience**, [S.l.], v.21, n.3, p.393–413, 2009.

MAGHRAOUI, K. E.; SZYMANSKI, B. K.; VARELA, C. A. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. In: PARALLEL PROCESSING AND APPLIED MATHEMATICS, 6TH INTERNATIONAL CONFERENCE, PPAM 2005, Poznan, Poland. **Anais...** Springer, 2006. p.258–271. (Lecture Notes in Computer Science, v.3911).

MATTSON, T. G.; SANDERS, B. A.; MASSINGILL, B. L. **Patterns for Parallel Computing**. [S.l.]: Addison Wesley, 2004. (Software Patterns Series).

MOR, S.; MAILLARD, N. Melhorando o Desempenho de Algoritmos do Tipo Branch & Bound em MPI via Escalonador com Roubo Aleatório de Tarefas. In: X SIMPÓSIO EM SISTEMAS COMPUTACIONAIS, São Paulo, Brazil. **Anais...** [S.l.: s.n.], 2009. p.11–18.

NIEUWPOORT, R. V. van et al. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. **J. of Supercomputing**, [S.l.], 2006.

NIEUWPOORT, R. V. van; KIELMANN, T.; BAL, H. E. Satin: efficient parallel divide-and-conquer in java. In: Euro-Par 2000 Parallel Processing, Munich, Germany. **Anais...** Springer, 2000. n.1900, p.690–699. (LNCS).

PEZZI, G. P. et al. Escalonamento Dinâmico de Programas MPI-2 Utilizando Divisão e Conquista. In: VII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, Ouro Preto, Brazil. **Anais...** [S.l.: s.n.], 2006. p.71–79.

PEZZI, G. P. et al. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD 2007), 19., Gramado - RS. **Anais...** IEEE Computer Society, 2007.

RAUBER, T.; RUNGER, G. A Data-Re-Distribution Library for Multi-Processor Task Programming. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS'05), 19., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.205a.

REINDERS, J. **Intel Threading Building Blocks**: outfitting c++ for multi-core processor parallelism. Sebastopol, USA: O'Reilly & Associates, Inc., 2007.

SUDARSAN, R.; RIBBENS, C. J.; FARKAS, D. Dynamic Resizing of Parallel Scientific Simulations: a case study using lammps. In: INTERNATIONAL CONFERENCE COMPUTATIONAL SCIENCE, PART I, 9. **Anais...** Springer, 2009. p.175–184. (Lecture Notes in Computer Science, v.5544).

UTRERA, G.; CORBALÁN, J.; LABARTA, J. Implementing Malleability on MPI Jobs. In: IEEE 13TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT 2004), Antibes Juan-les-Pins, France. **Anais...** IEEE Computer Society, 2004. p.215–224.

WEATHERLY, D. B. et al. Dyn-MPI: supporting mpi on medium-scale, non-dedicated clusters. **Journal of Parallel Distributed Computing**, Orlando, FL, USA, v.66, n.6, p.822–838, 2006.

WILSON, G.; IRVIN, R. **Assessing and comparing the usability of parallel programming systems**. Technical Report CSRI-321, University of Toronto, 1995.

ZHANG, G. et al. Scheduling and Data Redistribution Strategies on Tree Platforms. In: INTERNATIONAL JOINT CONFERENCE ONCOMPUTATIONAL SCIENCES AND OPTIMIZATION, 2009. CSO 2009. **Anais...** [S.l.: s.n.], 2009. v.1, p.105–108.

APPENDIX A – RESUMO ESTENDIDO

Este capítulo apresenta um resumo estendido, em português, do conteúdo abordado nesta tese. Cada uma das seções a seguir resumem os capítulos da tese expondo as principais considerações, ideias e contribuições.

A.1 Introdução

O Capítulo 1 expôs a motivação desta tese, a qual se refere a necessidade de se prover adaptabilidade às aplicações paralelas a fim de proporcionar um uso pleno das potencialidades das arquiteturas paralelas atuais. Esta tese esteve focada em dois níveis de adaptabilidade: maleabilidade - capacidade de adaptação, em tempo de execução, a variações na quantidade de processadores disponíveis; e evolutividade - capacidade de adaptação, em tempo de execução, a variações na carga de trabalho das aplicações que ocorrem sem que possam ser previstas. Nosso trabalho foi voltado a prover adaptabilidade a aplicações MPI (*Message-Passing Interface*). Esta interface foi escolhida por ser o padrão *de facto* para o desenvolvimento de aplicações de alto desempenho para arquiteturas de memória distribuída. Nossa proposta consistiu em tirar proveito da criação dinâmica de processos definida pelo padrão MPI-2 para introduzir uma certa flexibilidade às aplicações MPI, possibilitando assim que as aplicações possam adaptar-se a variações tanto na quantidade de processadores quanto na carga de trabalho. A hipótese considerada nesta tese é que o exploração das características do MPI-2, associado a implementação do suporte requerido no nível do ambiente de execução, torna possível a execução de aplicações MPI adaptativas nas arquiteturas paralelas atuais.

Parte I – Adaptabilidade: Contexto e Trabalhos Relacionados

A.2 Contexto: Adaptabilidade em Ambientes Paralelos

O Capítulo 2 apresentou conceitos, incluíndo a taxonomia adotada, e o contexto onde requer-se adaptabilidade em ambientes paralelos. Inicialmente apresentou-se a classificação de Feitelson e Rudolph, a qual considera quem decide o número de processadores necessários para computar a aplicação (usuário ou o sistema gerenciador de recursos) e quando esta decisão é tomada (em tempo de inicialização ou execução). Nela estão contemplados os jobs maleáveis e evolutivos, os quais representam os tipos de adaptabilidade almejados nesta tese. Após, apresentou-se cenários típicos e casos de uso de jobs maleáveis e evolutivos, demonstrando tanto características no nível da

aplicação para a implementação da adaptabilidade, quanto o suporte requerido no nível do ambiente de execução para que se possa prover adaptabilidade. Um levantamento dos requisitos necessários para viabilizar a maleabilidade (suporte a variações no número de processadores) e a evolutividade (suporte a variações imprevistas na carga de trabalho) foram apresentados na sequência. Neste último, o foco voltou-se para o paradigma de paralelismo de tarefas explícitas (*Explicit Task Paralellism*), o qual tem sido largamente utilizado como um modelo de extração do paralelismo em tempo de execução eficiente para arquiteturas multi-core. Por fim, analisou-se qual o impacto, em relação a estrutura de programa utilizada, de se prover um comportamento adaptativo à aplicações paralelas. Para isto, detalhou-se quatro estruturas de programa (SPMD - *Single Program, Multiple Data*, Mestre/Trabalhador, Laço Paralelo, *Fork/Join*) e como é possível implementar adaptabilidade nelas de acordo com a literatura.

A.3 Trabalhos Relacionados: Execução de Aplicações Adaptativas

No Capítulo 3 foram apresentados os trabalhos relacionados a esta tese. A análise foi dividida conforme três aspectos principais: *(i)* questões relativas ao suporte requerido do sistema gerenciador de recursos (RMS - *Resource Management System*) para suportar a alocação de um conjunto variável de processadores a uma aplicação maleável; *(ii)* APIs (*Application Programming Interface*) que permitem a implementação de aplicações que suportam e tratam de variações imprevistas na carga de trabalho de aplicações paralelas e *(iii)* políticas de escalonamento *on-line* a fim de proporcionar o balanceamento da carga de trabalho de aplicações adaptativas.

Parte II – Provendo Adaptabilidade à Aplicações MPI

A.4 Como Prover Adaptabilidade usando MPI?

O Capítulo 4 buscou responder a seguinte pergunta:"Como prover adaptabilidade usando MPI?". Nossa proposta consiste em tirar proveito da criação dinâmica de processos, definida no MPI-2, como meio para implementar adaptabilidade em aplicações MPI, respondendo assim ao questionamento. Inicialmente, o Capítulo 4 apresentadou as características básicas do MPI, como é possível criar novos processos MPI em tempo de execução, a relação de comunicação que passa a existir entre o processo que cria (processo pai) e o processo criado (filho) e uma breve análise do custo extra proveniente da criação de processos em tempo de execução. Após, discutiu-se as interações necessárias entre aplicações MPI maleáveis e o RMS a fim de permitir a execução deste tipo de aplicação em clusters de computadores. Conjuntamente, apresentou-se como é possível implementar aplicações MPI maleáveis utilizando a criação dinâmica de processos. Para as aplicações evolutivas almejadas, ou seja aplicações que seguem o paradigma de paralelismo de tarefas explícitas, apresentou-se como é possível implementar este tipo de comportamento através da criação dinâmica de processos e os requisitos necessários para a execução deste tipo de aplicação MPI. Exemplificando o desenvolvimento de aplicações MPI adaptativas, são apresentados pseudo-códigos MPI implementando maleabilidade e evolutividade para uma aplicação de multiplicação de matrizes.

A.5 Executando Aplicações MPI Maleáveis em Clusters

O Capítulo 5 descreveu como foi possível executar aplicações MPI maleáveis em clusters de computadores e os resultados obtidos. Primeiramente, apresentou-se o RMS utilizado, o OAR, o qual suporta o gerenciamento de um conjunto dinâmico de processadores e é capaz de interagir com aplicações MPI maleáveis para as devidas atualizações de informações. Após, são apresentados detalhes técnicos de diferentes distribuições MPI que implementam a criação dinâmica de processos. Todas as distribuições são idênticas quanto a funcionalidade provida (criação de processos em tempo de execução), entretanto elas diferem quanto a forma de se viabilizar alterações na quantidade de recursos utilizados. Adicionalmente, descreveu-se o escalonador desenvolvido para viabilizar a comunicação e o gereciamento das mensagens trocadas entre RMS e aplicação MPI maleável. Os resultados obtidos permitiram avaliar o desempenho obtido por uma aplicação MPI maleável. Foi possível observar ganhos de desempenho quando a aplicação expandiu-se a fim de fazer uso de novos processadores que foram disponibilizados em tempo de execução. Quando a aplicação necessitou reduzir-se para liberar processadores que foram solitados pelo RMS, a aplicação sofreu perdas de desempenho. Entretanto, é importante notar que a aplicação foi capaz de executar sem falhas sobre um conjunto variável de processadores. Aplicações com este tipo de comportamento são frequentemente utilizadas para aumentar a taxa de utilização dos processadores em clusters de computadores. Esta tese incluiu uma análise da utilização de um cluster executando job maleáveis nos seus processadores ociosos. O conjunto de processadores ociosos de um cluster varia constantemente conforme novos jobs vão sendo lançados e outros encerram sua execução. Comparou-se a utilização dos processadores obtida pelos nossos jobs MPI maleáveis com job MPI moldáveis, ou seja, jobs que em sua inicialização adaptam-se a quantidade de processadores disponibilizados sem que seja possível alterar esta quantidade em tempo de execução. Toda a execução de um job moldável é encerrada quando faz-se necessário liberar um ou mais processadores e este é incapaz de utilizar processadores além da quantidade definida em tempo de inicialização. No nosso cenário de teste, os jobs MPI maleáveis permitiram que fosse possível aumentar a utilização do cluster em mais de 25% quando comparado aos jobs moldáveis.

A.6 Paralelismo de Tarefas Explícitas em Aplicações MPI

O Capítulo 6 apresentou nossa proposta para o desenvolvimento e execução de aplicações MPI que sigam o paradigma de paralelismo de tarefas explícitas, as quais correspondem ao tipo de job evolutivo almejado nesta tese. Inicialmente, é feita a definição de uma tarefa abstrata em MPI, a qual representa uma funcionalidade ao invés de um processo, que é como geralmente as distribuições MPI implementam as tarefas MPI. Tarefas abstratas serão associadas a processos MPI (ou seja, associadas as tarefas MPI conforme a nomenclatura do padrão), logo, faz-se necessário utilizar estratégias a fim de garantir que as tarefas MPI possuirão granularidade suficiente para serem eficientes. A extração do paralelismo em tempo de execução deve ser coerente com o grau de paralelismo da arquitetura paralela alvo (por exemplo, o número de cores numa arquiteturas multi-core) e com os dados de entrada da aplicação. Devido a características inerentes do problema a ser solucionado, é preciso definir as definir as dependências entre as tarefas abstratas e as consequentes

transferências de dados necessárias para satisfazê-las. No caso das aplicações MPI, o meio convencional para solucionar dependências de dados é através de trocas de mensagens entre processos MPI, o qual será empregado nas aplicações MPI evolutivas propostas. As tarefas abstratas são extraídas em tempo de execução, em consequência, faz-se necessário mapeá-las nos processos MPI e garantir que haverá um bom balanceamento de carga dos processos. Nosso trabalho buscou uma distribuição igualitária de tarefas abstratas entre os processos e baseou-se numa boa distribuição de carga para ajustar a granularidade a fim de garantir o balanceamento de carga. Os resultados experimentais apresentados visaram validar a extração do paralelismo em aplicações MPI em tempo de execução; analisar o desempenho obtido por este tipo de aplicação MPI e verificar o impacto da estratégia proposta para controlar a granularidade dos processos MPI. Os resultados mostraram que é possível extrair o paralelismo em tempo de execução fazendo uso da criação dinâmica de processos e fazendo uso da troca de mensagens para resolver as dependências de dados. O desempenho obtido com as aplicações MPI evolutivas que foram propostas, foi comparável ao desempenho obtido por APIs que fornecem a extração do paralelismo explícito como o OpenMP. Adicionalmente, verificamos que a extração do paralelismo em tempo de execução em aplicações MPI com carga regulares atingem níveis de eficiência satisfatórios.

A.7 Conclusão

O Capítulo 7 expôs as considerações finais desta tese, a qual almejou prover e dar suporte a aplicações MPI adaptativas. A proposta deste trabalho baseou-se na flexibilização oferecida pela criação dinâmica de processos que foi utilizada para implementar ações adaptativas em aplicações MPI. Os estudos desta tese estiveram focados em dois aspectos:

- **Adaptar a execução de aplicações MPI a alterações na disponibilidade do seu conjunto de processadores.** Devido ao grande número de usuários de arquiteturas paralelas que competem por recursos, o número de processadores disponíveis tende a variar constantemente. Quando o conjunto de processadores utilizados por uma aplicação sofre alterações, para usufruir plenamente das potencialidades dos recursos, é necessário que a aplicação adapte-se a em tempo de execução, ou seja, que ela seja *maleável*. Esta adaptação envolve ações de expansão, para fazer uso de novos processadores, e de redução para liberar processadores que estão sendo solicitados pelo RMS. Estas ações devem ocorrer de maneira que a aplicação não falhe ou retorne resultados inconsistentes. A criação dinâmica de processos permite que a aplicação expanda-se, ou seja, crie novos processos para fazer uso de processadores adicionais;
- **Adaptar a extração do paralelismo em tempo de execução de acordo com a arquitetura paralela e os dados de entrada.** A difusão das arquiteturas multi-core requer que as aplicações paralelas possam adaptar-se a variados números de elementos de processamento (neste caso, diferentes números de cores). Neste contexto, o paralelismo de tarefas explícitas (*Explicit Task Parallelism*) é um importante paradigma de programação: ele extrai o paralelismo em tempo de execução conforme o grau de paralelismo da arquitetura

alvo e os dados de entrada. A criação dinâmica de processos permite implementar a extração do paralelismo: novos processos são criados sob demanda e suas dependências são solucionadas através da troca de mensagens.

Os aspectos descritos acima representam um novo uso da interface MPI. Assim, a primeira questão a ser tratada é como se desenvolve aplicações adaptativas com MPI. Esta tese apresentou a primitiva `MPI_Comm_spawn` que provê a criação de novos processos em tempo de execução, seu funcionamento e a relação hierárquica de comunicação resultante do uso desta primitiva. Adicionalmente, foram apresentados os requisitos para suportar maleabilidade em aplicações SPMD e Mestre/Trabalhador, assim como protótipos das rotinas requeridas. Basicamente são requeridos: `malleability_handle` para identificar as mudanças ocorridas no conjunto de processadores e lançar as ações adaptativas; `growth_action` que lança processos nos processadores disponibilizados depois do inicio da execução da aplicação, transferindo a eles alguma carga de trabalho e `shrinkage_action` que libera processadores finalizando apropriadamente seus processos garantindo assim a corretude da aplicação.

Os requisitos para suportar o paralelismo de tarefas explícitas em aplicações MPI também foram descritos nesta tese. Nosso foco foram as aplicações *Fork/Join* e algoritmos D&C, os quais são largamente utilizados pois cada chamada recursiva pode representar uma tarefa abstrata. Nossa proposta é substituir as chamadas recursivas por criações dinâmicas de processos, mapeando assim tarefas abstratas para tarefas MPI. Entretanto, um mapeamento direto entre tarefas abstratas e tarefas MPI leva a criação excessiva de novos processos. Esta tese buscou ajustar esta associação a fim de reduzir o impacto da criação de processos em tempo de execução. As sincronizações entre as tarefas abstratas (ou seja, retorno das funções recursivas) foram providos por trocas de mensagens no MPI. O ponto chave para obter desempenho neste tipo de aplicação está ligado ao escalonamento: mapear os processos criados dinamicamente entre os processadores disponíveis e ajustar a granularidade dos processos MPI para garantir um bom balanceamento de carga.

Exemplificou-se o uso do `MPI_Comm_spawn` como meio de suporte a adapatabilidade através de um exemplo prático de duas versões de uma multiplicação de matrizes: uma maleável e outa seguindo o paralelismo de tarefas explícitas. Descreveu-se as questões técnicas no desenvolvimento de aplicações MPI adaptativas. Para aplicações maleáveis é necessário prover meios de interações entre a aplicação e ao RMS para transmitir as atualização da disponibilidade dos processadores. Para aplicações que implementa o paralelismo de tarefas explícitas é necessário algum mecanismo para prover o controle da granularidade dos processos MPI.

Nos resultados desta tese foi utilizado o sistema gerenciador de recursos OAR, sob o qual implementamos sua integração com nossa proposta de aplicações MPI maleáveis. Este sistema possui um tipo especial de job, *Best Effort*, que permite implementar a alteração na quantidade de processadores disponíveis a uma aplicação. Adicionalmente, o OAR possui uma política de gerenciamento capaz de lidar com as demandas de jobs maleáveis. Os jobs maleáveis no OAR foram implementados da seguinte forma: novos processadores são adicionados ao conjunto de uma aplicação através do lançamento de novos jobs *Best Effort*; processadores são liberados em tempo de execução através da finalização de jobs *Best Effort* que executam nos processadores requeridos.

Nossos resultados experimentais mostraram que os jobs maleáveis proporcionaram um aumento de 25% na utilização dos processadores de um cluster quanto com-

parado com uma abordagem moldável. Este ganho foi medido em um cluster de produção: nós utilizamos a carga de trabalho de um cluster real, onde foram executados jobs rígidos, para carregar nosso ambiente de testes. Assim, foi possível repetir os testes com as mesmas condições para os jobs maleáveis e moldáveis, onde pode-se identificar o impacto do uso de cada um deles na utilização do cluster.

Nesta tese foi descrita nossa proposta de aplicações MPI seguindo o paradigma de paralelismo de tarefas explícitas, representando as aplicações evolutivas almejadas. Nele, foram definidas as tarefas MPI abstratas, as sincronizações necessárias para satisfazer as dependências entre elas (tirando proveito da comunicação eficiente provida pelo MPI) e o controle da granularidade como meio de garantir o balanceamento de carga destas aplicações. Foram apresentadas duas alternativas: uma no nível do ambiente de execução que é capaz de controlar a granularidade em tempo de execução sem requerer a intervenção do programador; e uma no nível do desenvolvimento da aplicação paralela onde o programador implementa o controle da granularidade.

Como a extração do paralelismo acontece em tempo de execução, faz-se necessário um escalonamento *on-line* para mapear os novos processos nos processadores disponíveis e balancear suas cargas de trabalho. O mapeamento é atendido pelas distribuições MPI que o fornecem por padrão. De acordo com a literatura, a estratégia mais eficiente para o escalonamento *on-line* é o *Work Stealing*. Nós descrevemos nesta tese as restrições que dificultam o oferecimento de um escalonar genérico baseado nesta estratégia. Como não foi possível oferecer uma versão genérica do *Work Stealing* para aplicações MPI com criação dinâmica de processos, nós optamos por controlar a granularidade das tarefas como meio de garantir o balanceamento da carga de trabalho em aplicações regulares.

Nossos resultados experimentais demonstraram que: *(i)* foi possível desenvolver aplicações MPI capazes de extrair o paralelismo em tempo de execução atingindo um desempenho similar ao OpenMP; *(ii)* foi possível executar eficientemente aplicações regulares implementando o paralelismo de tarefas explícitas em ambientes de memória distribuída; *(iii)* o mecanismo de ajuste da granularidade das tarefas MPI proposto possibilitou a adaptação de acordo com o grau de paralelismo da arquitetura alvo (multi-core) e os dados de entrada para aplicações regulares. Através destes resultados confirmamos nossa hipótese de que é possível obter eficiência na execução em memória distribuída de aplicações regulares implementadas de acordo com paradigma de paralelismo de tarefas explícitas. Entretanto, oferecer mecanismos para tornar transparente aos desenvolvedores os detalhes técnicos da extração dinâmica do paralelismo em aplicações MPI, da forma como propusemos nesta tese, ainda é um desafio em aberto.

Contribuições

As principais contribuições desta tese são:

- Determinar como as características do MPI-2 podem ser usadas para implementar aplicações MPI adaptativas;
- Uma proposta de projeto de aplicações MPI maleáveis;
- A proposta de um mecanismo de interação entre o OAR e as aplicações maleáveis propostas nesta tese;
- A proposta do projeto de aplicações MPI capazes de extrair o paralelismo em

tempo de execução;

Trabalhos Futuros e Perspectivas

Durante a elaboração desta tese, nós nos deparamos com muitos desafios tanto técnicos quanto de pesquisa. A seguir listamos algumas possibilidades de novas pesquisas identificadas através desta tese.

- Nós implementamos aplicações MPI maleáveis usando soluções simples para prover as ações adaptativas. Um possível trabalho futuro seria analisar o impacto do uso de outras soluções, como por exemplo, usar um mecanismo de *checkpointing-restart* para implementar as ações de redução. Assim, seria possível garantir que não haveria nenhuma perda de trabalho já computado. Outra possibilidade neste sentido seria fazer uso da migração de processos na implementação das ações adaptativas;
- O desenvolvimento de programas paralelos para arquiteturas multi-core é naturalmente multithreaded. Muitas pesquisas integram MPI e threads, por exemplo providas pelo OpenMP, em programas híbridos. As aplicações MPI adaptativas podem ser desenvolvidas de forma híbrida. Adicionalmente, estas aplicações podem ser projetadas para atender a outros tipos de arquiteturas, tais como as GP-GPUs, nas quais a adaptabilidade ajudaria a atingir eficiência;
- Nosso grupo de pesquisa tem estudado a adaptação da estratégia *Work Stealing* às características das aplicações MPI com criação dinâmica de processos. Pretende-se prover uma implementação genérica desta estratégia para tais aplicações. Logo, um trabalho futuro consiste na investigação de meios para contornar as restrições encontradas devido ao modelo de comunicação hierárquico;
- Consideranso programas híbridos (MPI e interface multithreaded), outra perspectiva de pesquisa seria restringir o uso do *Work Stealing* num escopo intra-nó (memória compartilhada), onde ele é provado como eficiente. Outra estratégia para distribuir a carga no escopo inter-nós (memória distribuída) poderia ser combinada para garantir o desempenho. Por exemplo, aplicar a estratégia *Adaptive*, descrita nesta tese, no escopo inter-nós associada ao *Work Stealing* no escopo intra-nó, pode permitir o suporte a aplicações adaptativas de forma transparente em clusters de computadores;
- Com relação ao paralelismo de tarefas explícitas em aplicações MPI, nossa pesquisa tem oportunidades para avançar. Por exemplo, ampliar a análise do desempenho, incluindo comparações com outras APIs como o KAAPI, e investir no desenvolvimento do controle da extração do paralelismo no nível do ambiente de execução, transparecendo ao programador as decisões de criação de novos processos ou de associação de tarefas abstratas.