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

RODRIGO DA ROSA RIGHI

# MigBSP: A New Approach for Processes Rescheduling Management on Bulk Synchronous Parallel Applications

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

Prof. Dr. Philippe Olivier Alexandre Navaux
Advisor

Prof. Dr. Hans-Ulrich Heiss
Sandwich Advisor

Porto Alegre, October 2009

**CIP – CATALOGING-IN-PUBLICATION**

Righi, Rodrigo da Rosa

MigBSP:
A New Approach for Processes Rescheduling Management on Bulk Synchronous Parallel Applications / Rodrigo da Rosa Righi. – Porto Alegre: PPGC da UFRGS, 2009.

141 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Advisor: Philippe Olivier Alexandre Navaux; Sandwich Advisor: Hans-Ulrich Heiss.

1. Communication. 2. Scheduling. 3. Load Balancing. 4. Bulk Synchronous Parallel. 5. Processes Migration. 6. Heterogeneity. 7. Dinamicity. I. Navaux, Philippe Olivier Alexandre. II. Heiss, Hans-Ulrich. III. Título.

*“The longest journey begins with the first step”*
— ANCIENT CHINESE PROVERB

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LISTA DE ALGORITHMS

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| AMPI | Adaptive MPI |
| BSP | Bulk Synchronous Parallel |
| Comm(i,j) | Communication Metric considering process *i* and Set *j* |
| Comp(i,j) | Computation Metric considering process *i* and Set *j* |
| DP | Dynamic programming |
| LAN | Local Area Network |
| Mem(i,j) | Memory Metric when analyzing process *i* and Set *j* |
| MPI | Message Passing Interface |
| MPMD | Multiple Program Multiple Data |
| PI(i) | Prediction of instruction of process *i* |
| PM(i,j) | Potential of Migration of the process *i* to Set *j* |
| SPMD | Single Program Multiple Data |
| WAN | Wide Area Network |

# ABSTRACT

This thesis treats the processes rescheduling problem during application runtime, offering dynamic load rebalancing among the available resources. Since most distributed computing scenarios involve more and more resources and dynamic applications, the load is a variable measure and an initial processes-processors deployment may not remain efficient with time. The resources and the network states can vary during application execution, as well as the amount of processing and the interactions among the processes. Consequently, the remapping of processes to new processors is pertinent to improve resource utilization and to minimize application execution time. In this context, this thesis presents a rescheduling model called MigBSP, which controls the processes migration of BSP (Bulk Synchronous Parallel) applications. BSP application model was adopted because it turns parallel programming easier and is very common in scientific applications development scenarios.

Considering the scope of BSP applications, the novel ideas of MigBSP are threefold: (i) combination of three metrics - Memory, Computation and Communication - in a scalar one in order to measure the potential of migration of each BSP process; (ii) employment of both Computation and Communication Patterns to control processes' regularity and; (iii) efficient adaptation regarding the periodicity to launch processes rescheduling. In our infrastructure, we are considering heterogeneous (different processor and network speed) distributed systems. The processes can pass messages among themselves and the parallel machine can gather local area networks and clusters. The proposed model provides a mathematical formalism to decide the following questions about load (BSP processes) balancing: (i) When to launch the processes rescheduling; (ii) Which processes will be candidates for migration and; (iii) Where to put the processes that will be migrated actually.

We used the simulation technique to validate MigBSP. Besides MigBSP, three scientific application were developed and executed using Simgrid simulator. In general, the results showed that MigBSP offers an opportunity to get performance in an effortless manner to the programmer since its does not need modification on application code. MigBSP makes possible gains of performance up to 20% as well as produces a low overhead when migrations do not take place . Its mean overhead is lower than 8% of the normal application execution time. This rate was obtained disabling any processes migration indicated by MigBSP. The results show that the union of considered metrics is a good solution to control processes migration. Moreover, they revealed that the developed adaptations are crucial to turn MigBSP execution viable, mainly on unbalanced environments.

**Keywords:** Communication, Scheduling, Load Balancing, Bulk Synchronous Parallel, Processes Migration, Heterogeneity, Dinamicity.

**MigBSP: Uma Nova Abordagem para o Gerenciamento de Reescalonamento de Processos em Aplicações Bulk Synchronous Parallel**

# RESUMO

A presente tese trata o problema do reescalonamento de processos durante a execução da aplicação, oferecendo rebalanceamento dinâmico de carga entre os recursos disponíveis. Uma vez que os cenários da computação distribuída envolvem cada vez mais recursos e aplicações dinâmicas, a carga é uma medida variável e um mapeamento inicial processos-recursos pode não permanecer eficiente no decorrer do tempo. O estado dos recursos e da rede podem variar no decorrer da aplicação, bem como a quantidade de processamento e a interação entre os processos. Consequentemente, o remapeamento de processos para novos recursos é pertinente para aumentar o uso dos recursos e minimizar o tempo de execução da aplicação. Nesse contexto, essa tese de doutorado apresenta um modelo de reescalonamento chamado MigBSP, o qual controla a migração de processos de aplicações BSP (*Bulk Synchronous Parallel*). O modelo de aplicação BSP foi adotado visto que torna a programação paralela mais fácil e é muito comum nos cenários de desenvolvimento de aplicações científicas.

Considerando o escopo de aplicações BSP, as novas idéias de MigBSP são em número de três: (i) combinação de três métricas - Memória, Computação e Comunicação - em uma outra escala com o intuito de medir o Potencial de Migração de cada processo BSP; (ii) emprego de um Padrão de Computação e outro Padrão de Comunicação para controlar a regularidade dos processos e; (iii) adatação eficiente na freqüência do lançamento do reescalonamento de processos. A infra-estrutura de máquina paralela considera sistemas distribuídos heterogêneos (diferentes velocidades de processador e de rede). Os processos podem passar mensagens entre si e a máquina paralela pode agregar redes locais e clusters. O modelo de reescalonamento provê um formalismo matemático para decidir as seguintes questões: (i) Quando lançar o reescalonamento dos processos; (ii) Quais processos são candidatos a migração e; (iii) Para onde os processos selecionados serão migrados.

A técnica de simulação foi usada para validar MigBSP. Além do próprio MigBSP, três aplicações científicas foram foram desenvolvidas e executadas usando o simulador Simgrid. Os resultados mostraram que MigBSP oferece oportunidade de ganhar desempenho sem alterações no código fonte da aplicação. MigBSP torna possível ganhos de desempenho na casa de 20%, bem como produz uma baixa sobrecarga quando migrações são inviáveis. Sua sobrecarga média ficou abaixo de 8% do tempo de execução normal da aplicação. Essa taxa foi obtida desabilitando quaisquer migrações indicadas por MigBSP. Os resultados mostraram que a união das métricas consideradas é uma boa solução para o controle de migração de processos. Além disso, eles revelaram que as adaptações desenvolvidas na freqüência do reescalonamento são cruciais para tornar a execução de MigBSP viável, principalmente em ambientes desbalanceados.

**Palavras-chave:** Escalonamento, migração de processos, *Bulk Synchronous Parallel*, desempenho, balanceamento de carga.

# 1 INTRODUCTION

Nowadays, the use of networks of computers is growing as platform to compute general-purpose problems. Especially in both resource sharing and high performance computing areas, we can observe the use of large and heterogeneous distributed systems, where issues like dynamicity, load balancing and resource management must be carefully analyzed (AUMAGE; HOFMAN; BAL, 2005; KANG et al., 2009). This growth in scale occurs because sometimes a single system such as a cluster or a computer does not offer a satisfactory performance or simply does not provide enough capacity of memory or disk storaging. Concerning this, an idea is to assemble heterogeneous parallel machines with a low financial cost, simply joining the power of clusters and workstations that belong to one or more institutions. Thus, grid networks (or just grids) arise to provide a distributed computational infrastructure for advanced science and engineering applications (BATISTA et al., 2008).

The use of parallel machines like grids crucially depends on the availability of a model of computation simple enough to provide a convenient basis for software development. In this context, a possibility is to use the Bulk-Synchronous Parallel (BSP) model of computation to write parallel programs for this infrastructure. This model has been proposed by L. G. Valiant as an unified framework for the design, analysis and programming of general purpose parallel computing systems (VALIANT, 1990). BSP applications are composed by a set of processes that execute supersteps. Each superstep is subdivided into three ordered phases: (i) local computations on each process; (ii) global communication actions using message passing and; (iii) a barrier synchronization.

Initially, BSP was created for homogeneous computing but may be used as an attractive model for heterogeneous and dynamic computing as well (CAMARGO; KON; GOLDMAN, 2005). BSP becomes attractive due to the simplicity to write parallel programs, because it is independent of target architectures and provides an idea of execution cost that combines its three phases. BSP has been used in a number of application areas, primarily in scientific computing (DOBBER; MEI; KOOLE, 2009). We can cite computational fluid dynamics applications such as those for solving 3D multi-grid viscous flows and to compute Lattice Boltzmann method. In addition, we have BSP applications to treat fast Fourier transform, quantum molecular dynamics and dynamic programming-based algorithms. BSP applications also treat with well-known problems in computer science like all sums, LU decomposition, sorting and optimal broadcast. Its usage on grid computing, for example, can be observed on the following works (VASILEV, 2003; CHEN; TONG, 2004; MARTIN; TISKIN, 2004; GOLDCHLEGER et al., 2005).

BSP model does not specify how the application's processes should be assigned to resources. The programmer/user must deal with scheduling issues in order to achieve better application efficiency. This issue is important since the barrier phase always wait for

the slowest process before starting the next superstep (BONORDEN, 2007). This topic is more relevant when grid environments are considered due to their own characteristics. Therefore, the task of allocating BSP processes to processors on such architecture often becomes a problem requiring considerable programmer/user effort. In order to fully exploit this kind of environment, he/she must know both the parallel machine architecture in advance and the BSP application code properly. Moreover, each new application requires another analysis for processes scheduling.

Since resource management and scheduling are key services for grid, issues like load balancing represent a common concern for most developers. A possibility is to explore the automatic load balancing at middleware level, linking it to the programming library. Considering the grain of work as a process of the operating system, an allocation scheme where the processes with longer computing times are mapped to faster machines can be used. This mechanism occurs before application execution or when its processes are created on the fly. However, this approach is not the best one for irregular applications and/or dynamic distributed environments, because a good process-resource assignment performed in the beginning of the application may not remain efficient with time (AGGARWAL; MOTWANI; ZHU, 2003; HUANG et al., 2006; LOW; LIU; SCHMIDT, 2007).

At the moment of processes launching, it is not possible sometimes to recognize either the amount of computation of each process or the communication patterns among them. Besides fluctuations in the processes' computing and communication actions, we have fluctuations related to the dynamicity presented in grids. In this kind of environment, processors' load may vary and networks can become congestioned while the application is running. Considering both the aspects of dynamicity (application and architecture), an alternative for obtaining performance is to apply processes rescheduling through their migration to new resources, offering application runtime load balancing (GALINDO; ALMEIDA; BADÍA-CONTELLES, 2008; CHEN; WANG; LAU, 2008).

## 1.1 Objectives

Scheduling schemes for grid systems can be viewed in two levels (FRACHTENBERG; SCHWIEGELSHOHN, 2008). In the first level processors are allocated to a job and in the second one, processes from a job are (re)scheduled using this pool of processors. Considering this classification, we developed a model called **MigBSP** that embraces this last scheme, offering algorithms for dynamic load (processes from the operating system) rebalancing among the resources during application runtime (ROSA RIGHI et al., 2008). Its parallel machine aims to join the power of clusters, LAN networks and multiprocessor machines. The main idea of the proposed model is to adjust the conclusion of both local computation and global communication phases of the BSP processes to be faster taking profit from information collected at runtime. This adjustment happens through **processes rescheduling**. Our main objectives for the development of MigBSP are described below.

- Employment of a model for parallel programming that comprises a large spectrum of scientific applications;

- Achieve better execution performance without changes in application's code and without previous knowledge about application's behavior;

- Consider pertinent data that impulse migration as well as those that act against it in order to choose the appropriate candidate processes for migration;

- Consider self-organizing and adaptations issues aiming to adjust the model impact on application execution on the fly.

Generally, dynamic load balancing algorithms are implemented directly with the application, resulting in the close coupling between the application and the load balancing algorithms data structure (CHAUBE; CARINO; BANICESCU, 2007). The task of use load balancing to allocate, or reallocate, processes to processors on grids often becomes yet a more complex problem for the programmer due to the dynamicity and heterogeneous features of the architecture. One solution was proposed by Bhandarkar, Brunner and Kale (2000) which developed a middleware for processes rescheduling. The load balancing is called by the application using a special directive MPI_Migrate(). This strategy imposes the fact that the programmer must know the best places inside the application for processes migration. Other possibility is to use automatic processes migration without extra code inside the application. In this way, our idea is to achieve performance without changing application's code directly, working at middleware level together with the programming library. Thus, we treat the BSP application as a black box that is linked to a middleware before its execution. In addition, we are working without previous knowledge about application behavior. All data about it is captured on the fly.

Frachtenberg and Schwiegelshohn (2008) affirm on their paper about new challenges of parallel job scheduling that there is a need to unify the current scheduling evaluation metrics in order to be useful and meaningful for the actual use cases. Considering the aspects of processes rescheduling, we will consider important data that influence this topic and combine them in order to create a new single metric to evaluate process migration. Firstly, we intend to capture data about processes behavior. In this context, both the computation and communication times of each process as well as the number of performed instructions and data transferred at each superstep will be considered. Furthermore, information regarding the parallel architecture must be used, such as the theoretical capacities and the loads of the processors and the available bandwidth to transfer data between two end points. Our spectrum of data also considers the migration costs. In this direction, the process' memory and the costs related to migration operations must be evaluated.

Nowadays, we can observe that there are approaches that consider only the computation aspect (MORENO-VOZMEDIANO; ALONSO-CONDE, 2005; BONORDEN, 2007; SALEHI; DELDARI, 2006; WANG et al., 2007), while others work only with communication data (KRIVOKAPIC; ISLINGER; KEMPER, 2000) in their migration algorithms. Other proposes include one of the aspects above together with migration costs (VADHIYAR; DONGARRA, 2005a; CHEN; WANG; LAU, 2008). In addition, we observe that there are some works that consider both computation and communication aspects, but not treat the migration costs (BHANDARKAR; BRUNNER; KALE, 2000; LOW, 2002). Our final idea is to combine data about computation and communication phases of the BSP supersteps, data about parallel machine and information regarding migration costs in order to create a scalar metric that informs the potential of migration of each BSP process. Thus, the adjustment performed by MigBSP happens through the migration of those processes which have long computation time, perform several communication actions with other processes that belong to a same network and present a low migration cost.

We observed that some works present fixed parameters. For example, Vadhiyar and Dongarra computed the migration cost as a fixed value (VADHIYAR; DONGARRA, 2005b). In addition, Bonorden (2007) presents the migration call for BSP processes at each superstep. Considering that the final objective is to minimize the application execution time, the adaptivity is a key idea for the model to self-optimize scheduling parameters. This self-optimization is important for two reasons: (i) achieve better performance in migrations; (ii) to generate as low impact from MigBSP in application execution as possible. Firstly, we work with up to date information in order to measure this cost more precisely on the fly. Other branch of adaptivity is the management of the interval for rescheduling launching. The frequency of the processes rescheduling calls will depend on both the system state (processes are balanced or not among the resources) and the result of past calls (if they generate processes migration or not). Other branch comprises the work with two adaptable patterns that act on computation and communication phases of each process, respectively. Their ideas are to measure the regularity of a process regarding the instructions that it executes and the amount of data that it communicates to other processes. We have established that processes' regularity is important for us because we have expected that a process performs with a similar behavior on the destination place.

## 1.2 Problem Statement

The abstract goal of our rescheduling model can be stated as follows:

*Given a BSP application and a set of processors on which BSP processes may be executed, the model must control the remapping of processes to processors in order to reduce the supersteps' times.*

Our final objective is to reduce the application execution time. For that, we will work measuring both the computation and communication times on every process at each superstep. These measurements will serve as a basis for decision-making about the rescheduling launching as well as to decide about which processes are candidates for migration. In order to achieve our statement above, the model also must cause less intrusion on application execution as possible. Since migrations may not take place under some conditions, such as high latency among the destination and source processors or a high amount of data is linked to a specific process, the impact of scheduling messages and computation from the model must be minimized.

## 1.3 Approach

A practical view of our work can be seen in Algorithm 1. MigBSP provides a mathematical formalism that answers the following issues regarding processes migration: (i) "When" to launch the processes migration; (ii) "Which" processes are candidates for migration and; (iii) "Where" to put an elected process from the candidates ones. We are not interested in the keyword "How", that treats the mechanism used to perform migrations.

Algorithm 1 presents an overview of the processes rescheduling function. This function may be called several times during application execution and its triggering occurs according to system feedback. Lines 1 and 2 are responsible to answer the question "Which". We will analyze the potential of migration of each BSP process based on three metrics: (i) Computation; (ii) Communication and; (iii) Memory. This last metric enters

**Algorithm 1** Processes rescheduling function

```
1: Evaluation of Computation, Communication and Memory Metrics
2: Selecting the candidate processes for migration
3: for each candidate process p do
4:     Find a destination processor P
5:     if profitable to load balance then
6:         Migration of process p to processor P
7:     end if
8: end for
```

in the model as an idea of migration cost and acts against processes transferring. After that, a mechanism will be applied to choose those processes that are candidates for migration. For each candidate, it is chosen a destination processor and its migration profitability to this place is evaluated (line 5). This operation takes into account the external load on source and destination processors, the simulation of considered process running in the destination processor, the BSP processes that both processors are executing, as well as the migration cost of considered process. Finally, for each candidate process is chosen a new resource or its migration is canceled. Thus, the operations on lines 4 up to 6 are responsible to answer the question "Where".

## 1.4 Document Organization

This document if organized in six chapters. After the introduction chapter, we present some information regarding the concepts covered in this thesis. Chapter 2 describes some topics such as scheduling, load balancing, processes migration, grid computing and the BSP execution model. Chapter 3 presents the state of the art of the systems and algorithms that perform scheduling, processes migration and load balancing. This chapter also deals with related work to the BSP model. Its main contribution is the definition of the gaps that is lacking work to obtain performance on this kind of parallel programming model.

Chapter 5 presents our model for processes rescheduling on BSP applications. This model is the main focus of this thesis and, then, this chapter is its most important part. We will describe in this chapter our algorithms that control BSP processes migration, as well as both the models of application and machine used in the thesis. Chapter 5 shows the evaluation of the developed model. Intermediaries chapters have a special section called "Summary" which makes a brief analysis of the thesis' part in evidence. Finally, we show Chapter 6 at the end of this document. It makes a conclusion about the text, emphasizing the MigBSP's main contributions and results.

# 2 FUNDAMENTALS

This chapter has an explanatory character, presenting some terminologies and concepts that will be used on the next chapters of this document. This chapter shows the relevance of processes rescheduling and load balancing in parallel and distributed processing areas. In special, this chapter covers these topics over heterogeneous and dynamic environments. Besides this, this chapter describes the BSP execution model, which will be employed as a model of application in our model for processes rescheduling.

The present chapter is segmented in six sections. Section 2.1 treats about scheduling. It presents a widely used taxonomy on this area and introduces the problem of rescheduling. Section 2.2 shows the importance of load balancing and some metrics that may be considered in this topic. Section 2.3 treats the migration problem technique and the costs involved on it. In addition, this section will address the issue about how the migration is triggered and its implementation regarding the programmer's point of view. Section 2.4 presents some concepts about grid computing. Section 2.5 describes the model for parallel programming called BSP (Bulk Synchronous Parallel). This chapter also shows the places where we can aggregate load-balancing algorithms on BSP applications in order to achieve better performance. Finally, Section 2.6 presents an abstract of this chapter, making a relation among the main topics discussed on it.

## 2.1 Scheduling

The scheduling problem, in a general view, comprises both a set of resources and a set of a consumers as illustrated in Figure 2.1 (THAIN; TANNENBAUM; LIVNY, 2005). This problem considers the finding of an efficient politic to manage the use of the resources by several consumers in order to optimize the performance metric chosen as parameter (YAMIN, 2001). Commonly, a scheduling proposal is evaluated by two features: (i) performance and; (ii) efficiency. Concerning this, the evaluation comprises the obtained scheduling as well as the time spent to execute the scheduler policies. For example, if the parameter to analyze the achieved scheduling is the application execution time, the lower this value the better the scheduler performance. On the other hand, the efficiency refers to the policies adopted by the scheduler and can be evaluated using computational complexity functions (TSENG; CHIN; WANG, 2009).

The general scheduling problem is the unification of two terms in common use in the literature. There is often an implicit distinction between the terms scheduling and allocation. Nevertheless, it can be argued that these are merely alternative formulations of the same problem, with allocation posed in terms of resource allocation (from the resources point of view), and scheduling viewed from the consumers point of view. In this sense, allocation and scheduling are merely two terms describing the same general mechanism,


Figure 2.1: General vision of the scheduling problem (THAIN; TANNENBAUM; LIVNY, 2005)

but described from different viewpoints (CASAVANT; KUHL, 1988). One important issue when treating scheduling is the grain of the consumers. For example, we can have a graph of tasks, a set of processes and jobs that need resources to execute (SCHNEIDER et al., 2009). In this context, scheduling schemes for multi-programmed parallel systems can be viewed in two levels (FRACHTENBERG; SCHWIEGELSHOHN, 2008). In the first level processors are allocated to a specific job. In the second level processes from a job are scheduled using this pool of processors.

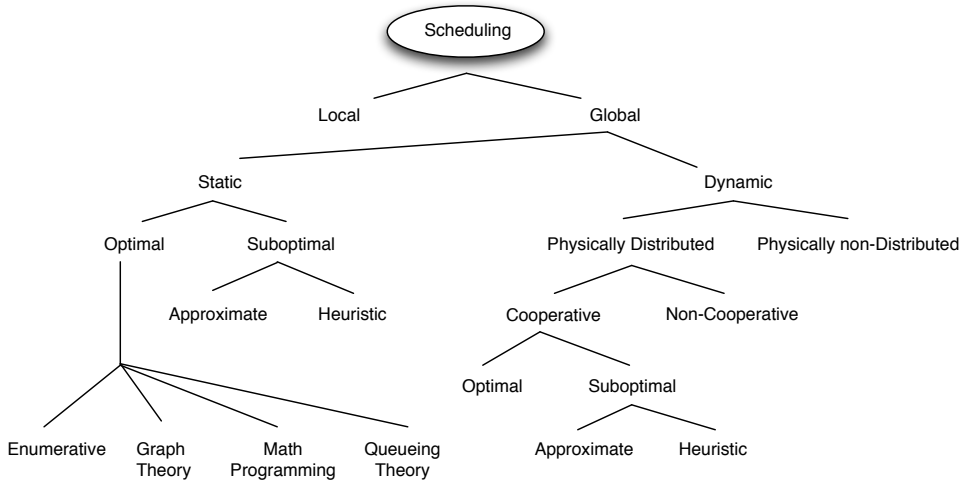


Figure 2.2: Scheduling taxonomy proposed by Casavant and Kuhl (1988)

In order to formalize the classification of schedulers, Casavant and Kuhl (1988) proposed a scheduling taxonomy for general purpose distributed computing systems. Figure 2.2 depicts such taxonomy proposed by these authors. The first division comprises either the local or global scheduling. Local scheduling is involved with the assignment of processes to the time-slices of a single processor. Global scheduling is the problem of deciding where to execute a process, while the job of local scheduling is left to the operating system of the processor in which the process is ultimately allocated. Global scheduling can be divided in two approaches, which are denoted below.

- Static;
- Dynamic.

We will define static scheduling considering the scheduling grain as a task. If data such as information about the processors, the execution time of the tasks, the size of the data, the communication pattern and the dependency relation among the tasks are known in advance, we can affirm that we have a static or deterministic scheduling model (TOPCUOUGLU; HARIRI; WU, 2002; BEAUMONT; ROBERT, 2002; LU; SUBRATA; ZOMAYA,

2006). In this approach, each executable image in the system has a static assignment to a particular set of processors. Scheduling decisions are made deterministically or probabilistically at compile time and remain constant during runtime. The static approach is simple to be implemented. However, Lu at al. (2006) pointed out that it has two major disadvantages. Firstly, the workload distribution and the behavior of many applications cannot be predicted before program execution. Secondly, static scheduling assumes that the characteristics of the computing resources and communication network are all known in advance and remain constant. Such an assumption may not be applied on grid environments, for instance (see Section 2.4).

In the general form of a static task scheduling problem, an application is represented by a directed acyclic graph (DAG) in which nodes represent application tasks and edges represent intertask data dependencies (TOPCUOUGLU; HARIRI; WU, 2002). Each node label shows computation cost (expected computation time) of the task and each edge label shows intertask communication cost (expected communication time) between tasks. The objective function of this problem is to map tasks onto processors and order their executions so that task-precedence requirements are satisfied and a minimum overall completion time is obtained.

In the case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal assignment can be proposed. Even with all information required for the scheduling, the static method is often computationally expensive getting to the point of being infeasible. Thus, this fact results in suboptimal solutions. We have two general categories within the realm of suboptimal solutions for the scheduling problem: (i) approximate and; (ii) heuristic. Approximate scheduling uses the same methods used in the optimal one, but instead exploring all possible ideal solutions it stops when a good one is achieved. Heuristic scheduling uses common parameters and ideas that affect the behavior of the parallel system (MARTÍNEZ-GALLAR; ALMEIDA; GIMÉNEZ, 2007). For example, we can group processes with higher communication rate to the same local network or to sort works and processors in lists following some predefined criteria in order to perform an efficient mapping among them (list scheduling) (JIN; SCHIAVONE; TURGUT, 2008).

Dynamic scheduling works with the idea that a little (or none) a priori knowledge about the needs and the behavior of the application is available (LI; LAN, 2004; LU; SUBRATA; ZOMAYA, 2006; WANG et al., 2007; WARAICH, 2008). It is also unknown in what environment the process will execute during its lifetime. The arrival of new tasks, the relation among them and data about the target architecture are unpredictable and the decision of the consumers-resources mapping is taken by the runtime environment. Following the interpretation of Figure 2.2, the responsibility of the global scheduling can be assigned either to a single processor (physically non-distributed) or practiced by a set of processors (physically distributed). Within the realm of this last classification, the taxonomy may also distinguish between those mechanisms that involve cooperation between the distributed components (cooperative) and those in which the individual processors make decisions independent of the actions of the other processors (non cooperative). In the cooperative case, each processor has the responsibility to carry out its own portion of the scheduling, but all processors are working toward a common system-wide goal.

Casavant and Kuhl (1988) also presents a horizontal classification for scheduling. In this context, the scheduling can be viewed as adaptive and non-adaptive. An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to the previous and the

current behavior of the system. According to Shah et al. (2007), adaptive algorithms are a special type of dynamic algorithms, where the scheduling parameters/options are optimized based on the global state of the system. An example of such an adaptive scheduler would be one which takes many parameters into consideration when making its decisions. In response to the behavior of the system, the scheduler may start to ignore one parameter or reduce its importance if it believes that parameter is either providing information that is inconsistent or it is not pertinent to a specific situation. Regarding the structure, Krauter et al. (2002) also proposed their scheduling taxonomy which is divided in three ways.

- Centralized;
- Hierarchical;
- Decentralized.

A centralized organization is characterized by the presence of only one scheduling controller. This controller is responsible for the system-wide decision-making. Such an organization has some advantages including easy management and simple deployment. Its disadvantages include the lack of both scalability and fault-tolerance. In addition, centralized schemes usually entail a global synchronization to obtain global information about the participants (LI; LAN, 2004). In a hierarchical organization, the scheduling controllers are organized in a hierarchical fashion. One obvious way of organizing the controllers would be to let the higher level controllers manage larger sets of resources and lower level controllers manage smaller sets of resources. Compared with the centralized scheduling, the hierarchical strategy addresses the scalability and fault-tolerance issues.

The decentralized organization is another alternative. It naturally addresses several important issues such as fault-tolerance, scalability, site-autonomy and multi-policy scheduling. Nevertheless, decentralized organizations introduce several problems, such as management and usage tracking. This scheme is expected to scale well to large network sizes but it is necessary for the scheduling controllers to coordinate with each other via some form of resource discovery or resource trading protocols. The overhead of operation of these protocols will be the determining factor for the scalability of the overall system. Lack of such protocols may reduce the overhead but the efficiency of scheduling may also decrease. In addition, other disadvantage of this approach is that local schedulers may take conflicting decisions regarding the use of the resources, which turn the global scheduling system inefficient.

The scheduling grain treated by the model described in this thesis is a process of the operating system. The scheduling of a process treats its initial mapping to a specific resource. Normally, this procedure is performed either when a new process is created during application runtime or when the application is launched. The scheduling problem can be extended and used to redistribute a process during its execution. This situation characterizes the problem known as rescheduling (HAO et al., 2008). Rescheduling can be viewed as a technique that reallocates consumers (processes, for example) that are already assigned to specific set of resources to execute (or continue their executions) in another one (AGGARWAL; MOTWANI; ZHU, 2003). Rescheduling is pertinent in dynamic environments like grids, where data obtained previously may not be confirmed during application runtime. Besides this, this technique is important to treat irregular applications in which their computation and/or communication behaviors change during the execution. Processes rescheduling may be implemented using processes migration techniques and commonly uses load balancing mechanisms on its policies.

## 2.2 Load Balancing

The basic idea of load balancing is to attempt to balance the load on all processors in such a way to allow all processes on all nodes to proceed at approximately with the same rate (CASAVANT; KUHL, 1988; WARAICH, 2008). The greatest reason to launch the load balancing is the fact that exists an amount of processing power that is not used efficiently, mainly in dynamic and heterogeneous environments like grids (YAGOUBI; MEDEBBER, 2007). In this context, schedulers' policies can use load balancing mechanisms for several purposes, such as: (i) to choose the amount of work to be sent to a process; (ii) to move work from an overloaded processor to another that presents a light load; (iii) to choose a place (node) to launch a new process from a parallel application and; (iv) to decide about processes migration. Load balancing is especially important for some parallel applications that present synchronization points, in which the processes must execute together the next step. Following this scenario, Figures 2.3 (a) and (b) show execution scenarios without and with load balancing, respectively. The situation (b) of Figure 2.3 may be obtained, for example, through the transferring of heavy processes to lightly loaded resources or through the work distribution according to processors' capacity and load.

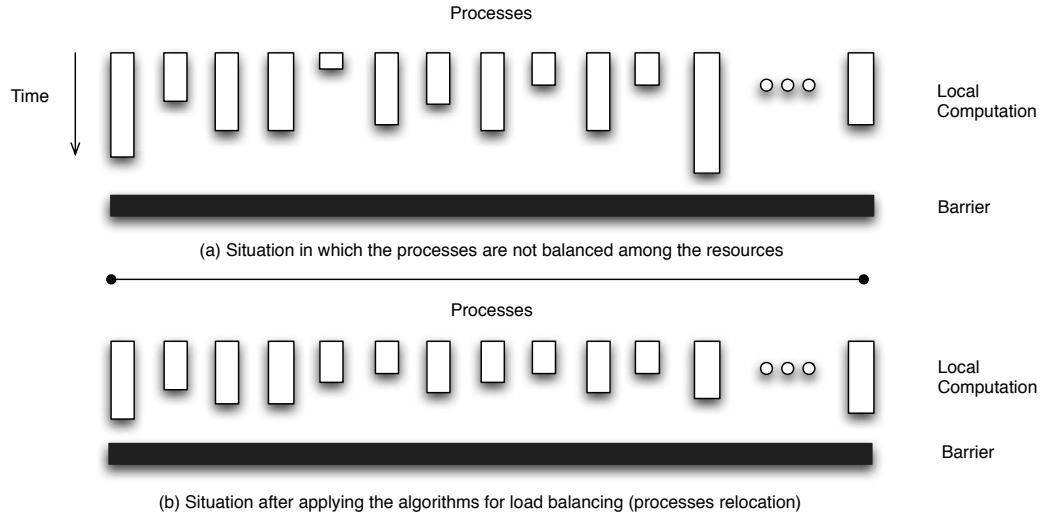


Figure 2.3: Example of situations without and with applying load-balancing algorithms

The most fundamental topic in load balancing consists in determining the measure of the load. There are many different possible measures of load including: (i) number of tasks in a queue; (ii) CPU load average; (iii) CPU utilization at specific moment; (iv) I/O utilization; (v) amount of free CPU; (vi) amount of free memory; (vii) amount of communication among the processes and so on. Besides this, we can have any combinations of the above indicators. Considering the scope of processes from the operating system, such measures will influence in deciding about when to trigger the load balancing, which processes will be involved and where are the destination places in which these processes will execute. Especially on the last topic, other factors to consider when selecting where to put a process include the nearness to resources, some processor and operating system capabilities, and specialized hardware/software features (LU; SUBRATA; ZOMAYA, 2006).

We must firstly determine when to balance the load to turn the mechanism useful. Doing so is comprised of two phases: (i) detecting that a load unbalancing exists and; (ii) determining if the cost of load balancing exceeds its possible benefits. Following Watts

and Taylor (WATTS; TAYLOR, 1998), load imbalance can be detected in two manners:

- Synchronous load-balancing detection;
- Asynchronous load-balancing detection.

Most scientific codes have inherent synchronization points. These barrier operations provide a natural and clean point in which load balancing can be launched. When a barrier is initiated, the average load of all the computers may be determined. If the aggregate efficiency is below some user-specified limit (threshold), the workload is considered as unbalanced. This approach explains the functioning of the synchronous load balancing detection. A load unbalancing can also be detected in an asynchronous fashion. A task/process can keep track of a window of load history. If its percentage utilization during this period drops below a critical threshold, it can create a global request for load balancing. Even if a load imbalance exists, it may be better not to load balance simply because the cost of its execution would exceed the benefits of a better work distribution. Therefore, the specification and the measurement of the costs must be carefully analyzed when defining a load-balancing algorithm in order to perform only productive modifications.

Scheduling policies use load-balancing techniques in order to achieve better results. In this manner, it is trivial that we have both the static and dynamic classifications for load balancing (WARAICH, 2008; LI; LAN, 2004). While static load balancing is performed with data known at compile time, dynamic one takes profit from updated data collected on application runtime. This last approach can deal with the dynamism feature inherent to grid environments as well as present in some irregular applications (ROSA RIGHI et al., 2009a). Waraich (2008) affirm that there are three major parameters which usually define the strategy that a specific load balancing algorithm will employ. Following him, these parameters answer three important questions denoted below:

- Who makes the load-balancing decision;
- Which information is used to make the load balancing decision;
- Where is the load balancing made.

The question about who makes the load balancing decision is answered based on whether a sender-initiated or receiver-initiated policy is employed. Both policies are used to share the load from heavily loaded processors to those in which the load is more moderate. In a sender-initiated policy, congestioned nodes attempt to move work to lightly loaded nodes. On the other hand, lightly loaded nodes look for heavily loaded ones from which work may be received in receiver-initiated approaches. An example of receiver-initiated approach is the random stealing, where a node attempts to steal work from a randomly selected node when it finds its own task queue empty (ZHANG et al., 2005). Considering the idea of processes rescheduling, a lightly loaded processor could send messages to other processors indicating that it could handle more work. Thus, an overloaded processor can send a process to this lightly loaded processor, contributing to minimize the overall time for application execution.

Global and local strategies answer the question about what information will be used to make the load balancing decision. In global policies, the load balancer uses the performance profiles of all available processes. In local policies, processes are partitioned

into different groups. Although local information exchange strategies may yield to less communication costs, global information exchange strategies tend to give more accurate decisions. The choice of a local or a global policy depends on the behavior that an application will exhibit. For global schemes, balanced load convergence is faster compared to a local scheme since all the processes are considered at the same time. However, this requires additional communication and synchronization among them (WARAICH, 2008). Centralized and distributed strategies answer the issue related to where the load balancing is made. In a centralized scheme, the load balancer is located on the master process and all decisions are made there. In a distributed scheme, the load balancer is replicated on all processes.

Following Kwok and Cheung (2004), load balancing algorithms can be analyzed according to a framework with four dimensions: (i) location policy; (ii) information policy; (iii) transfer policy and; (iv) selection policy. These policies are explained in Table 2.1. In a nutshell, a load balancing system must develop these four policies that include the definition of the processing grain (tasks or processes, for example), the machines that may be considered to execute this grain, the load information and how the balancing mechanism will be activated.

Table 2.1: Four dimensions of a framework for load balancing (KWOK; CHEUNG, 2004)

| Dimension | Responsibilities |
|---|---|
| Location Policy | The objective of this policy is to find a suitable transfer partner for a machine, once the transfer policy has decided that the machine is a heavily-loaded state or lightly-loaded one. Common location policies include: random selection, dynamic selection and state polling. |
| Information Policy | This policy determines when the information about the state of other machines should be collected, from where it has to be collected and what information is to be collected. Common approaches are: no exchange of states, state probing (or demand-driven) in process of load balancing, periodic exchange (information gathered periodically), state-change broadcasting and conditional multicasting. |
| Transfer Policy | A transfer policy determines whether a machine is in a suitable state to participate in a task transfer, either as a sender or a receiver. For example, a heavily loaded machine could try to start process migration when its load index exceeds a certain threshold. |
| Selection Policy | This policy determines which task should be transferred. Once the transfer policy decides that a machine is in a heavily loaded state, the selection policy selects a task for transferring. Selection policies can be categorized into two policies: (i) preemptive and; (ii) non-preemptive. A preemptive policy selects a partially executed task. As such, a preemptive policy should also transfer the task state, which can be very large or complex. Normally, this transferring operation is expensive. A non-preemptive policy selects only tasks that did not begin their executions. |

Load balancing can be offered directly inside the application code. This approach results in a close coupling between the load balancing data structures and the application. Thus, such an implementation cannot be used with other applications owing to the specificity of the data structure shared both by the application and the algorithm (CHAUBE; CARINO; BANICESCU, 2007). Even more, sometimes the programmer may not have

expertise in optimizing the partitioning algorithm and the implementation of the load balancing algorithms takes time away from the developers' primary interest (the application). Other possibility is to offer load balancing at middleware level. In this context, a load-balancing module can be implemented together with the programming library. Considering the programmer's point of view, this approach represents an effortless manner to achieve performance. For example, load-balancing module for BSP processes rescheduling can be implemented directly in programming library. Communication data collected by communication directives and data about processors' capacity and load can be used for decision making of processes relocation. This decision may be taken at the barrier synchronization function, since this point means a pertinent moment to implement processes migration.

## 2.3 Processes Migration

A process is an abstraction of the operating system and represents an instance of a program in execution. Process migration is the movement of a currently executing process to a new processor. Migration offers an illusion of computation movement when, in reality, a new thread or process is being created at the destination where an equivalent state and environment will be reconstructed and resumed (MILANéS; RODRIGUEZ; SCHULZE, 2008). Figure 2.4 illustrates the execution flow of a migrated process. The transferring of the process' state implies on migration costs, which must be considered in viability calculus when a process is tested for migration (NEVES et al., 2007). Processes migration can be employed for processes rescheduling, allowing the system to take advantage of changes in processes behavior and/or in network utilization during processes execution. Thus, dynamic changes on the topology of the application can help in load balancing or in minimizing the downtime of non-interruptible services. Furthermore, scheduling algorithms gain more flexibility since the initial processes-resources assignment can be modified at any moment.

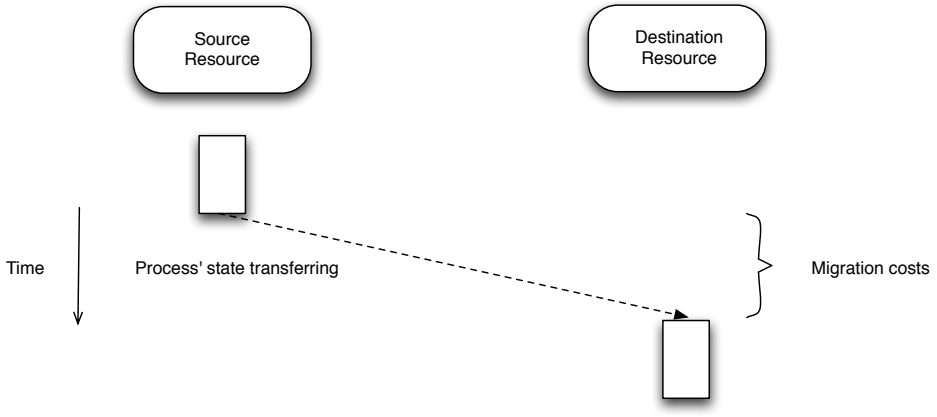


Figure 2.4: Process migration representation (MILOJICIC et al., 2000)

As a general view, processes migration may be used for the following issues: (i) dynamic load redistribution (KOVACS; KACSUK, 2004; ZHANG et al., 2005); (ii) high throughput computation, taking profit from idle computation cycles (BONORDEN; GEHWEILER; HEIDE, 2005); (iii) fault tolerance (GEHWEILER; SCHOMAKER, 2006; NAGARAJAN et al., 2007; WANG et al., 2008); (iv) improve

data access locality (JENKS; GAUDIOT, 2002) and; (v) for system administration (BEGNUM, 2007). Moving the computation closer to the data or to special resources or services enhances locality (item iv related above). For instance, if a process is accessing a huge remote data source, it may be more interesting to move it close to the source of data instead to move the data up to the machine where this process executes. This results in lesser communication and delay and makes more efficient the use of network resources. Locality issue is also relevant when there are networks with different capacities. The procedure to approximate two processes that have high communication rate to the same network level tends do minimize the time spent on their interactions.

Processes migration is also important to develop self-management systems. An example in this context is the Condor system (THAIN; TANNENBAUM; LIVNY, 2005). In this system, each computer belongs to a specific user and is available only when its owner does not use it. Condor migrates a process when the owner of the computer returns. The migration concept is also employed in autonomous objects transferring over wide area networks (KRIVOKAPIC; ISLINGER; KEMPER, 2000). Each object is independent and decides when and where it must migrate in this idea. There are 3 important components common to any processes migration policy (BOUKERCHE; DAS, 1997):

- Load balancing policy;
- Process migration mechanism;
- Remote execution.

The load balancing policy decides “when” and “where” to transfer a process, referenced by the keyword “Which”. Process migration mechanism decides “how” the system migrates a process from the source to the destination. Remote execution is responsible for the process execution on a new machine. The part of the migration policy component addressed in this thesis is the load balancing. The decision of when is important since it does not make much sense to perform load balancing if a task is only going to execute for a short period of time. Load balancing is only useful for long-running tasks with large CPU and I/O requirements. Besides the answer for the keyword when, our model for BSP processes rescheduling answers the following issues: (i) which processes are candidates for migration and; (ii) where to put an elected process from the candidates ones. We do not have interest in the keyword How, that treats the mechanisms used to implement the processes migration.

Migration is said to be transparent when the effects of the movement are hidden from the user and the application. In addition, migration can be initiated either from inside the computation (proactive or subjective migration) or from outside (reactive, objective, or forced migration) (MILANéS; RODRIGUEZ; SCHULZE, 2008). While subjective migration is more frequently found (especially in mobile agent applications), it affects the clean separation between application and mobility logic. On the other hand, various issues arise in objective migration. Care should be taken not to interrupt the computation while the system presents an inconsistent state. While interrupting the computation in the subjective case is trivial, it is a complex issue for the objective case. Independently of the used approach, careful placement of migration points is mandatory to avoid an excessive latency in responding to migration requests. Having too many points may lead to space and time overhead. In general, it is accepted as enough to place the migration points at the beginning of loops and function calls (MILANéS; RODRIGUEZ; SCHULZE, 2008).

## 2.4 Grid Computing

The grid is a concept of an infrastructure that intends to make use of widespread, hence loosely-coupled, large scale computational network and resources (CASANOVA, 2002). Its purpose is to superspass limitations on geographical locations and hardware specifications of computers. One of the promises of grid computing is to enable the execution of applications across multiple sites. These applications can be bag-of-tasks, workflows or parallel applications based on message passing paradigm (NETTO; BUYYA, 2008; SCHNEIDER et al., 2009). In this way, grid platform has been used for executing computing-intensive and data-intensive applications such as bioinformatics, high energy physics, climate modeling, economics, automotive/aerospace industry, and so forth (CHEN et al., 2008). The term "grid" was introduced in the book "The Grid: Blueprint of a New Computing Infrastructure", written by Ian Foster (FOSTER; KESSELMAN, 2003). Foster explains it in analogy to the electrical power grid, which is a singular entity that connects billions of power outlets and supplies a vast amount of heterogeneous devices with energy. This was the starting point of a whole new research field that distinguishes from conventional distributed computing.

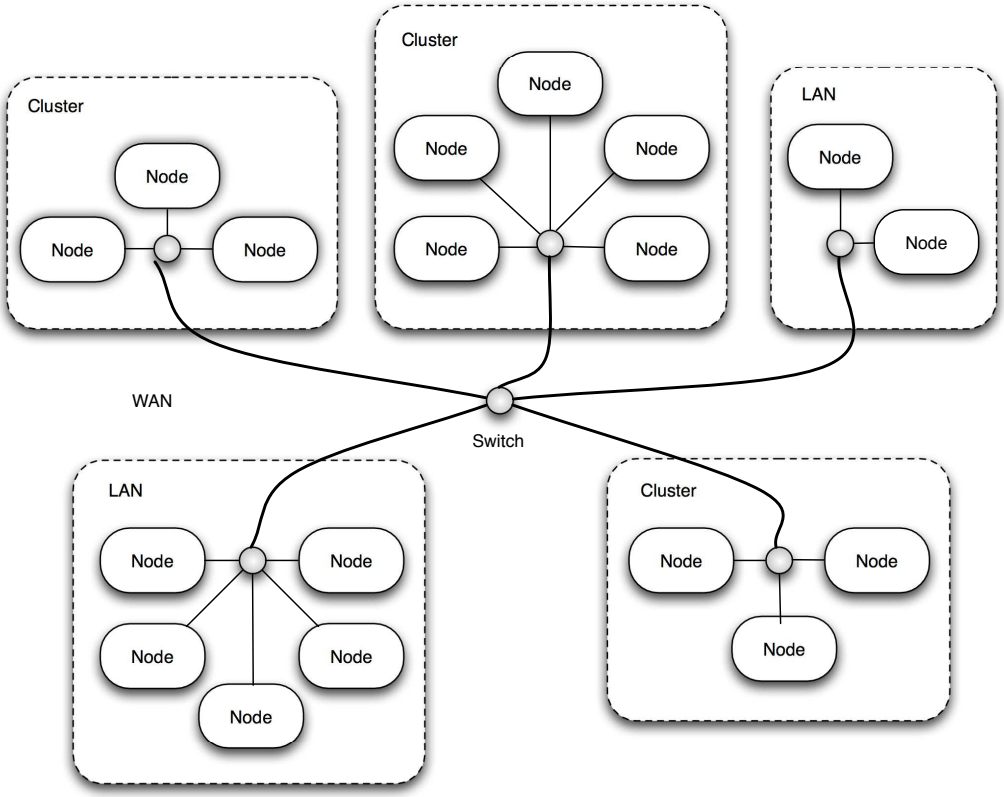


Figure 2.5: Example of a grid topology

Cluster computing incorporates a lot of properties that describe a grid. A cluster is a couple of resources to form a virtual singular resource. Usually a middleware manages job execution and usage of computational nodes. But a cluster is mostly associated with one organization and is not widespread, like the grid. Cluster nodes are more homogeneous, meaning they have at least all the same programming model and have mostly all the same hardware. In addition, clusters are normally static and have a high focus on topology, in terms of high speed, sophisticated interconnection networks and the way the nodes are

connected (e.g. torus, tree, hypercube and so on). As computational grids emerged and got widely used, resources of multiple clusters became dominant computing nodes of the grid. Cross-cluster process migration can help us to balance the load among multiple grid nodes with fine granularity (WANG et al., 2007). From the topological point of view, we can regard a grid computing as a set of clusters in a multi-nodes platform (YAGOUBI; MEDEBBER, 2007). Each cluster owns a set of worker nodes and belongs to a local domain, i.e. a LAN (Local Area Network). Every cluster is connected to the global network or WAN (World Area Network) by a switch. This idea of grid topology is illustrated in Figure 2.5.

The emergence of a variety of new applications demands that grids support efficient data, scheduling and resource management mechanisms (KRAUTER; BUYYA; MAHESWARAN, 2002). Yu and Shi consider that the static scheduling is not the best option for grid environments (YU; SHI, 2007). Static scheduling does not consider the future change of grid environment after the resource mapping is made. Most static scheduling approaches assume that resource set is given and fixed over time. The assumption is not always valid even with the reservation capability in place. Moreover, the static scheduling approach cannot utilize new resources after the plan is made and cannot reorganize the processes-processors mapping.

## 2.5 Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model was introduced by Leslie Valiant in 1990 as a model that joins both parts of machine architecture and software architecture (VALIANT, 1990). Bulk Synchronous Parallelism can be seen as a style of parallel programming developed for general-purpose parallelism, which is parallelism across all application areas and a wide range of architectures. BSP's most fundamental properties are(SKILLICORN; HILL; MCCOLL, 1997):

- It is simple to write;
- It is independent of target architectures;
- The performance of a program on a given architecture is predictable.

BSP programs look much the same as sequential programs. Only a bare minimum of extra information needs to be supplied to describe the use of parallelism. Unlike many parallel-programming systems, BSP is designed to be architecture-independent. Thus, programs run unchanged when they are moved from one architecture to another. The execution time of a BSP program can be computed from the text of the program and a few simple parameters of the target architecture. This makes design possible, since the effect of a decision on performance can be determined at the time it is made (VASILEV, 2003; NICULESCU, 2006). The BSP model comprises a collection of processors, each one with its own local memory, and a global communication network. Figure 2.6 depicts a BSP parallel machine. It is important to emphasize that there is not restriction either about processors proximity or about the type of the network (local or wide area network). BSP model just requires all-to-all and asynchronous communication among the processes.

In terms of BSP programs, we can affirm that they have both a vertical structure and a horizontal structure. The vertical structure arises from the progress of a computation through time. Considering the BSP scope, this structure is a sequential composition of

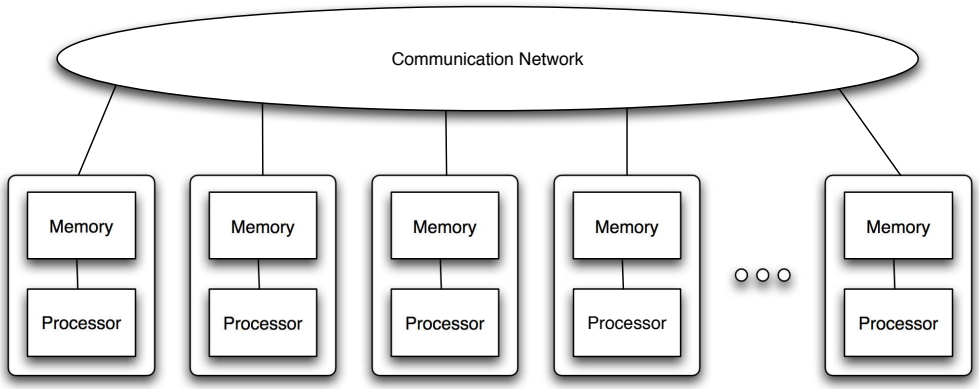


Figure 2.6: Simplified vision of a BSP parallel machine (SKILLICORN; HILL; MCCOLL, 1997)

global supersteps, which conceptually occupy the full width of the executing architecture. Thus, a BSP application is composed by one or more supersteps. Each superstep is further subdivided into three ordered phases which will be explained below and illustrated in Figure 2.7.

- Local computation in each process, using only values stored in the memory of its processor;
- Communication actions among the processes, involving movement of data between processors;
- A barrier synchronization, which waits for completing all communication actions.

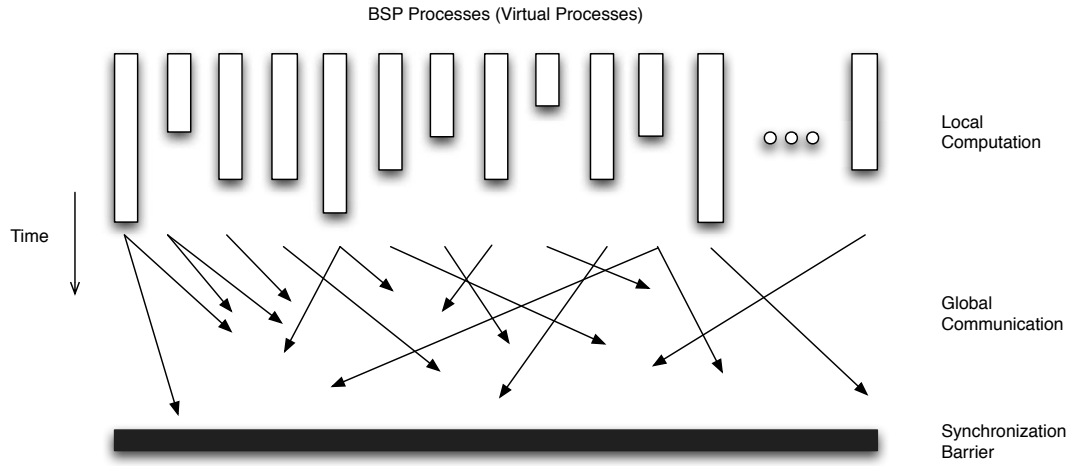


Figure 2.7: Representation of a BSP superstep in BSP model (SKILLICORN; HILL; MCCOLL, 1997)

The postponing of communications to the end of a superstep is the key idea for implementations of the BSP model (DUTOT et al., 2005). It removes the need to support nonbarrier synchronizations among processes and guarantees that processes within a superstep are mutually independent. This makes BSP easier to implement on different architectures and makes BSP programs easier to write, to understand, and to analyze mathematically.

The use of a barrier by the BSP model, although potentially costly, presents a number of attractive features. There is no possibility of deadlock or livelock in a BSP program because barriers make circularities in data dependencies impossible. Therefore, there is no need for tools to detect and deal with them. Barriers also permit novel forms of fault tolerance, because there is a forced synchronization in communication actions(SKILLICORN; HILL; MCCOLL, 1997; WANG et al., 2008). It is possible to achieve a global state applying checkpoints on each processes at this phase. Taking into account that a BSP application is a sequence of supersteps, if any fault occurs in a certain superstep, the application needs to rollback for only one superstep. This superstep mechanism makes the model to be reconfigured easily (MIAO; TONG, 2007). In addition, the barrier moment allows the easier increment or decrement of processes during application runtime, depending on the resources availability in the distributed system (GOLDCHLEGER et al., 2005). Consequently, this feature can be seen as a good point to create dynamic BSP applications, allowing some kind of variation in the number of processes.

The horizontal structure arises from concurrency and consists of a fixed number of virtual processes (BSP processes). These processes are not regarded as having a particular linear order and may be mapped to processors in any way. Thus, locality plays no role in the first placement of processes on processors. The performance of a BSP application can be characterized by three parameters: (i) $p$ = number of processors; (ii) $l$ = time spent to perform the barrier synchronization and; (iii) $g$ = rate in which the data accessed in a random fashion may be delivered. The parameter $g$ is related to the bisection bandwidth of the communication network and depends on some factors such as: (i) the protocols used for network communication; (ii) the buffer management by both the processors and the communication network; (iii) the routing strategy used in the communication network and; (iv) the BSP runtime system. Summarizing, the cost to start a data transferring at each superstep is added to $g$ value.

The separation of the local computation, global communication and synchronization phases permits to calculate the limit times and to predict the application performance just using simple equations. The cost a superstep can be determined as follows. Supposing that the work $w$ is the cost of local computations on each process during superstep $s$, $h_{out}$ as the amount of messages sent by any process during superstep $s$, $h_{in}$ as a maximum amount of messages received by each process during superstep $s$ and $l$ as the barrier synchronization cost. The time to compute the superstep $s$ can be calculated through Equation 2.1.

$$superstep\ cost\ =\ MAXw\ +\ MAX\{h_{in}, h_{out}\}\ .\ g\ +\ l \tag{2.1}$$

$$application\ cost\ =\ \sum_{s=0}^{s} W_s\ +\ g\ .\ \sum_{s=0}^{s} H_s\ +\ s\ .\ l \tag{2.2}$$

Equation 2.1 can be simplified just adopting $W$ and $H$ as the maximum times for $MAXw$ and $MAXh$. The total cost of a BSP application is the summing of the partial costs obtained at each superstep (see Equation 2.2). In Equation 2.2, the variable $s$ represents the number of supersteps. The cost model also shows how to predict performance across target architectures. The values of $p$, $w$, and $h$ for each superstep, and the number of supersteps can be determined by inspection of the program code. Values of $g$, and $l$ can then be inserted into the cost formula to estimate execution time before the program is executed. Therefore, the cost model can be used as part of the design process for BSP programs and to predict the performance of programs ported to new parallel computers.

Moreover, this idea of cost is useful to guide buying decisions for parallel computers if the BSP program characteristics of typical workloads are known.

The existence of a cost model that is both tractable and accurate makes it possible to truly design BSP programs. Concerning this, Table 2.2 presents some initiatives to improve the performance of a BSP application.

Table 2.2: Mechanisms to improve the performance of BSP applications

| Mechanism | Description |
|---|---|
| Balance the computation | Balance the computation among processes at each superstep, since the value of $w$ is the maximum over computation times and the barrier synchronization must wait for the slowest process. Considering a dynamic and heterogeneous parallel architecture, we can also reschedule the processes among the available resources taking into account their values of $w$. |
| Balance the communication | Balance the communication among the processes, since $h$ is a maximum over fan-in and fan-out of data. This communication equilibrium is pertinent when there are communications that use congestioned links or the Internet. We can remap the processes in order to approximate to the same network level those processes that present a higher communication pattern. |
| Reduce supersteps | Minimize the number of supersteps, since this determines the number of times $l$ appears in the final cost. |

BSP model can be expressed in a wide variety of programming languages and systems. For example, BSP programs could be written using existing communication libraries such as PVM (SUNDERAM, 1990) and MPI (DONGARRA et al., 1995). A programming library must offer just a barrier synchronization function and asynchronous communication directives. Compared to PVM and MPI, BSP approach offers the following facilities (SKILLICORN; HILL; MCCOLL, 1997): (i) a simple programming discipline (based on supersteps) that determines the correctness of programs easier; (ii) a cost model for performance analysis and prediction which is simple and compositional and; (iii) more efficient implementations on many machines. The most common approach for BSP programming is SPMD (Simple Program Multiple Data) (WILKINSON; ALLEN, 1999) imperative programming using Fortran or C languages. In addition, BSP programs can use traditional directives for message passing (send and receive) as well as remote direct memory access (RDMA) mechanisms. Section 3.4 presents some BSP communication libraries developed for cluster and grid computing.

## 2.6 Summary

In this chapter we presented a short description of the scheduling, load balancing and migration topics. Scheduling mechanisms manage the interaction between consumers and resources. Consumers represent the scheduling grain, which may be represented by a task, a process or a job, for example. A scheduler acts according to its scheduling policy, which can offer load balancing in order to maintain all the consumers in the same (or approximate) execution rate. Methods for load balancing can be classified into two categories: (i) static and; (ii) dynamic. Static approaches are usually achieved by task/process allocation beforehand. It requires prior knowledge of tasks/processes (such as execution time) and

cannot adapt to the runtime load variation. Dynamic approaches are more complex, since the data capturing and the procedure itself are done at runtime. Since static approaches usually cannot cope with the dynamic changes on system conditions, dynamic algorithms are used in practical situations. However, dynamic algorithms must collect, store, and analyze state information, and thus, they incur more overhead than their static counterparts.

Considering the scheduling grain as a process of the operating system, normally its scheduling to a resource happens before application execution characterizing a static approach. The procedure to map processes to resources before application execution does not consider possible modifications on infrastructure. Thus, a pertinent idea is to use dynamic processes creation taking into account data collect during application runtime. However, even using a dynamic approach, modifications after process launching occurred by either environment or application dynamicity may turn the previous scheduling inefficient in time. In this context, the fact to propose a new scheduling taking into account a previous one arises and brings the idea of rescheduling.

The support of processes rescheduling made with processes migration is one of the most important techniques to fully utilize the resources of the entire system. This mechanism can be managed by a dynamic load balancer, which constantly verifies the situation of both the infrastructure and the application and offers a processes-processors reorganization in order to maintain a specific level of performance. The rescheduling facilitates the dynamic load distribution and can improve the performance of both the communication and computation parts of the application. The computation time can be optimized through the transferring of processes from overloaded processors to others with moderated load. The communication can be improved minimizing the costs for message passing among the processes. It is possible to bring to the same network two processes that present a higher communication pattern or when an Internet link is observed. In addition, we must observe the costs for process state transferring when load balancing takes place. This must be carefully analyzed because sometimes the amount of process memory and/or the state of the network are issues that turn processes migration inviable. Thus, migration is useful only in cases where the gain achieved by migration outweighs the incurred cost.

BSP execution model was chosen to develop the algorithms of our processes rescheduling model. BSP model is being demonstrated as a successful paradigm for offering both predictable performance and architecture independent software. It certainly would be beneficial to transplant the features of BSP model to grid environment so as to create a programming model for grid and for performance predictable applications (MIAO; TONG, 2007). BSP presents two points of optimization that will be addressed by MigBSP: (i) in local computations and: (ii) in global communications. Both points can be improved in a heterogeneous (in terms of processing capacity and network velocity) and dynamic (use of processing resources and fluctuations on network bandwidth) distributed systems. An idea to make viable this improvement in BSP programs is the employment of processes rescheduling technique together with dynamic load balancing. Processes rescheduling can contribute to minimize the application execution time and maximizes the use of the resources. A trivial alternative to launch the rescheduling on BSP programs could be inside the barrier synchronization function, since it represents a point in which a consistent global state is achieved. Moreover, the barrier phase means an easier point to implement the mechanism for processes migration.

# 3 RELATED WORK

This chapter describes some related work that are inserted in the context of the current thesis document. It shows different systems and algorithms, as well as their approaches to deal with the following topics: scheduling, load balancing and processes migration. Furthermore, the present chapter addresses some works related to the BSP model and grid computing. Lastly, we present a special section that emphasizes the existing gaps inside the considered context and guides the reader to the next (and the main) chapter of this thesis.

The present chapter is organized in 4 sections. It is important to notice that this organization was made just to simplify and clarify the reading of the chapter. We will present systems and ideas that could be enclosed in more than one section. Section 3.1 treats the scheduling and load balancing topics. Section 3.2 shows some systems that deal with rescheduling and migration. Section 3.3 shows some related works in the context of grid computing. Section 3.4 describes some initiatives of BSP systems as well as some changes and improvements made on such model. Section 3.5 closes the chapter with a discussion of the main points addressed on it.

## 3.1 Scheduling

Elsasser, Monien and Preis (ELSASSER; MONIEN; PREIS, 2000) consider the load balancing topic on heterogeneous environments. The load must be balanced among the processors proportionally to their given power. In other words, given a network with $n$ nodes where each node $i$ contains a work load of $w_i$ and a weight of $c_i$, calculate a load balancing flow over the edges of the network such that, after termination, each node $i$ has the balanced proportional work load $\overline{w_i}$. $\overline{w_i}$ is computed following Equation3.1.

$$\overline{w_i} = \frac{\sum_{i=1}^{n} w_i}{\sum_{i=1}^{n} c_i} \cdot c_i \tag{3.1}$$

If the global imbalance vector $w - \overline{w}$ is known, it is possible to solve the problem by solving a linear system of equations. If only local information about the load imbalance must be used, the nodes of the graph have to balance the load with their neighbors iteratively until the whole network is globally balanced (ELSASSER; MONIEN; PREIS, 2000). This kind of algorithm is called diffusion. Diffusion algorithms assume that a node of the graph can send and receive messages to/from all its neighbors simultaneously.

Load balancing techniques are also employed in information dissemination systems like Publish/Subscribe (FIEGE et al., 2006). Cheung and Jacobsen (2006) work with dynamic load balancing in distributed content-based Publish/Subscribe systems. These systems may suffer from performance degradation and poor scalability under uneven load

distributions typical in real-world applications. Formerly, a broker can be overloaded if the incoming message rate into it exceeds the processing/matching rate supported by the matching engine. Secondly, the overloading can also occur if the output transmission rate exceeds the total available output bandwidth. Hence, Cheung and Jacobsen developed a load balancing algorithm that distributes the load by offloading subscribers from heavily loaded brokers to less loaded brokers.

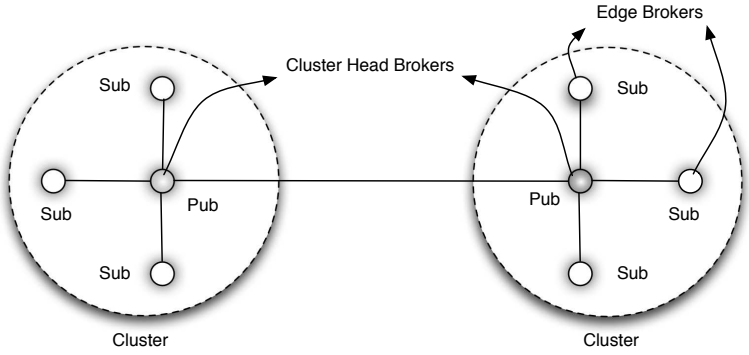


Figure 3.1: Publish/Subscribe architecture designed with clusters and managers (CHEUNG; JACOBSEN, 2006)

Cheung and Jacobsen developed an architecture called PEER (Padres Efficient Event Routing), in which organizes the brokers into a hierarchical structure as illustrates Figure 3.1. Brokers with more than one neighboring broker are referred to as cluster-head brokers, while brokers with only one neighbor are referred to as edge brokers. A cluster-head broker with its connected set of edge brokers forms a cluster. Brokers within a cluster are assumed to be closer to each other in network proximity. PEER's organization of brokers into clusters allows two levels of load balancing: (i) local-level (referred as local load balancing) where edge brokers within the same cluster load balance with each other and; (ii) global-level (referred as global load balancing) where edge brokers from two different clusters balance the load with each other. Edge brokers only need to exchange load information with edge brokers in the same cluster, and neighboring clusters can exchange aggregated load information about their own edge brokers.

Brokers must exchange load information with each other in order to know when and which brokers are available for load balancing. With this data, a detection algorithm can then trigger load balancing whenever it detects an overload or a wide load difference with another broker. The load unbalancing detection runs periodically. Detection allows a broker/cluster to monitor its current resource usage and also compare it with neighboring brokers/clusters so that it can invoke load balancing if necessary. The local detection algorithm running on an edge broker is composed of two steps. The first one identifies whether the broker itself is overloaded by examining four utilization ratios, namely: (i) input; (ii) output; (iii) CPU and; (iv) memory utilization. In this point, the parameter lower overload threshold is introduced to prevent the broker from accepting further load when one of its utilization ratios exceeds 0.9. If an utilization ratio exceeds the higher overload threshold at 0.95, then load balancing is invoked immediately.

Legrand et al. (2003) present a load balancing of iterative computations on heterogeneous clusters with shared communication links. These authors modeled the computation and communication costs of an application executed in phases. At each iteration, some independent calculations carry out in parallel by each process and then some communications take place. This organization remembers the functioning of the BSP model.

Kondo et al. (2002) described a client-server scheduling model for global computing. The application model consists in work units and result units. A client receives work units, processes them and returns result units. The computation platform is composed by a central server and a set of clients. A client is defined with characteristics like disk, CPU and network connection with the server. In addition, a client just executes some work when its CPU is idle. The model measures the processor speed ($Max_{cpu}$), the network bandwidth ($Max_{net}$) and the disk space ($Max_{disk}$) of each client in order to determine the number of work units that the server will pass to it. These three values are passed to work units. They are not combined and the minimum of them gives the amount of work that the server will pass to a specific client.

Yagoubi and Medebber (2007) proposed a load balancing model for grid environments. They propose a tree-based model to represent grid architecture in order to manage the workload. The model is characterized by three main features: (i) it is hierarchical; (ii) it supports heterogeneity and scalability and; (iii) it is totally independent from any grid physical architecture. In addition, The main characteristics of the proposed load balancing strategy are: (i) it uses a task-level load balancing; (ii) it privileges local tasks transfer to reduce communication costs and; (iii) it is a distributed strategy with local decision making. The load balancing strategy works with the idea that a grid is composed by a set of clusters or LANs (Local Area Networks). It works with three levels of management as we can see in Figure 3.2. In the first level (level 0), a Grid Manager decides to start a global load balancing between the clusters of the grid. In level 1, each Cluster Manager decides to start local load balancing. Finally, worker nodes are presented in level 2. They maintain their workload information and send this data to their own Cluster Managers.

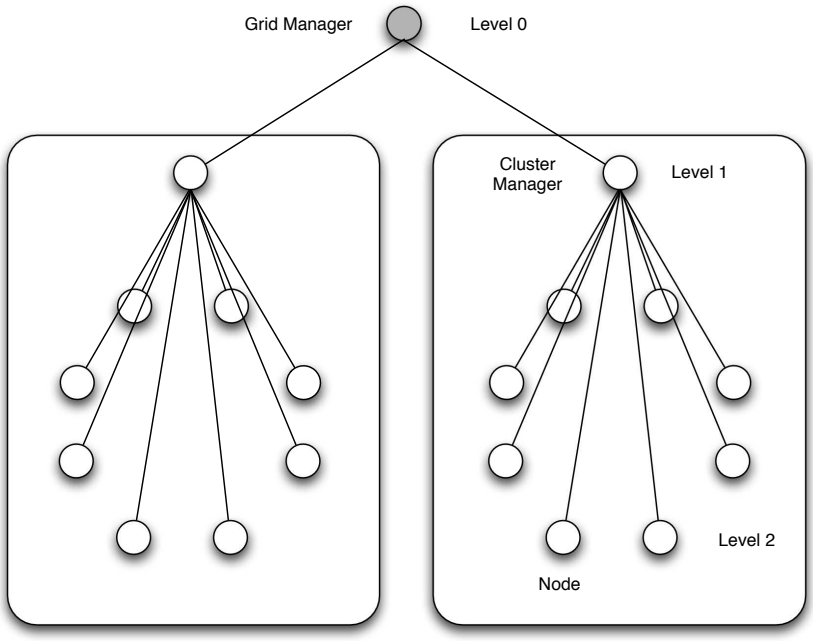


Figure 3.2: Tree-based representation of a grid (YAGOUBI; MEDEBBER, 2007)

In accordance with the hierarchical structure, the Yagoubi and Medebber proposed two load balancing levels: intra-cluster (or inter-nodes) and intra-grid (or inter-clusters). The intra-grid load balancing is performed only if some cluster Manager fails to load balance its workload among their associated worker nodes. The local balancing failure may be due to either saturation of the cluster or insufficient supply. The main advantage

of this strategy is to privilege local load balancing in first (within a cluster and then on the entire grid) (YAGOUBI; MEDEBBER, 2007). The goal of this neighborhood approach is to decrease the amount of messages exchanged between clusters. As a consequence, the communication overhead induced by tasks transfer and flow information is reduced.

Orduna et al. (2000) developed a communication-aware task scheduling system for heterogeneous systems. The interconnection network in heterogeneous systems may become the system bottleneck, particularly when executing applications with huge network bandwidth requirements, like multimedia and on-demand applications. Concerning this, they developed a model of communication cost between nodes. They proposed a simple metric that is the equivalent distance between each pair of nodes. This metric measures the cost of communicating two nodes without explicitly considering traffic pattern. The application of this method produces a table Tn of $N$ x $N$ equivalent distances, where $N$ is the number of nodes in the network.

Garcia and Morales-Luna (GARCÍA; MORALES-LUNA, 2008) designed a model for task assignment based on the notion of force. The Force-Field task assignment method takes both communication cost and idle cycles into consideration and combines them into a scalar force field. The net attraction of each remote node for a particular task will thus be the strength or affinity for receiving a particular task. This attraction in the force field will determine which remote computers are assigned the tasks that compose the superstep. Remote computers which do not have enough resources will produce a repulsive force.

Let the term computational ease refer to the reciprocal of the cost in cycles required to complete a task. What distinguishes the Force-Field task assignment method is, in general terms, the fact that force is in inverse ratio to the square of the cost of the communications and direct ratio to the computational resources at the remote node multiplied with the computational ease of the task to be assigned. The equation which determines the Force-Field is obtained through an analogy to Coulomb's law, as we can see in Equation 3.2.

$$F = \frac{q_j.M_{i,j}}{r_{ij}^2} \tag{3.2}$$

The remote computer charge is represented by $q_j$. This variable is the computational potential of node $j$ and is equivalent to the available computational cycles per unit time. This number is always positive or equal to zero. The task charge is denoted by $M_{i,j}$. This variable is the computational ease associated to the memory requirements of task $i$ with regard to node $j$. This variable will be less than zero, unless the memory resources at node $j$ are insufficient to satisfy the requirements of task $i$. The processor distance is represented by $r_{i,j}$ in Equation 3.2. $r_{i,j}$ is the cost in communications from the remote node $i$ to the local node $j$ from where the task assignment will take place. If $F < 0$ , there is a tendency to move a task towards the remote node, while $F > 0$ implies repulsion. With the Force-Field task assignment, the nodes with the most negative forces will be assigned the tasks: the assignment of tasks to nodes is done simultaneously in such a manner that the node with the greatest attractive force will receive the task associated with this force.

Topcuoglu et al. (2002) propose two new static scheduling algorithms for a bounded number of fully connected heterogeneous processors. The first algorithm is called Heterogeneous Earliest-Finish-Time (HEFT) while the second one refers to the Critical-Path-on-a-Processor (CPOP). Although the static scheduling for heterogeneous systems is offline, in order to provide a practical solution, the scheduling time (or running time) of an algorithm is the key constraint. Therefore, the motivation behind these algorithms is to deliver good-quality scheduling (or outputs with better scheduling lengths) with lower costs (i.e.,

lower scheduling times). The HEFT Algorithm selects the task with the highest upward rank at each step. The selected task is then assigned to the processor which minimizes its earliest finish time with an insertion-based approach. The upward rank of a task is the length of the critical path (the longest path) from the task to an exit task, including the computation cost of the task. The CPOP algorithm selects the task with the highest (upward rank + downward rank) value at each step. The algorithm targets scheduling of all critical tasks (tasks on the critical path of the DAG) onto a single processor, which minimizes the total execution time of the critical tasks.

## 3.2 Rescheduling and Migration

Bhandarkar, Brunner and Kale (2000) developed a framework to treat the load balancing in applications modeled as a collection of computing objects which communicate with each other. This framework is illustrated in Figure 3.3. The main component of the framework is the distributed database, which coordinates load balancing activities. Whenever a method of a particular object runs, the time consumed by that object is recorded. Furthermore, whenever objects communicate, the database records information about the communication which permit the construction of the object-communication graph. The proposed framework uses the Charm++ (ZHENG; HUANG; KALé, 2006) tool which offers several advantages for load balancing. Firstly, parallel programs are composed of many coarse-grained objects, which represent convenient units of work for migration. In addition, the CPU time and the message track for each object can be captured and analyzed by the framework at any particular time.

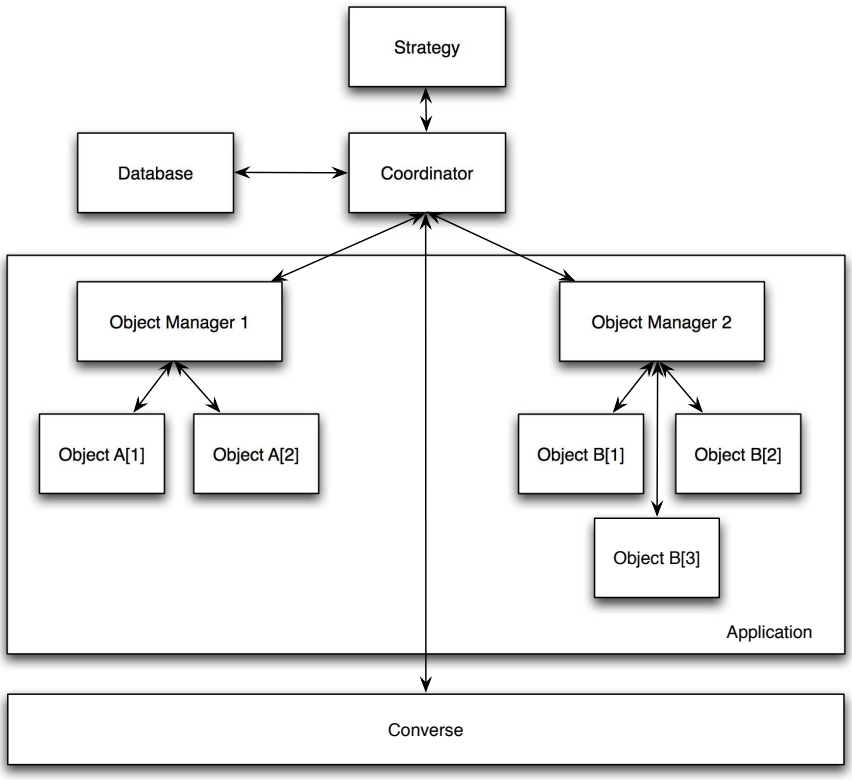


Figure 3.3: Components of the load balancing framework on a processor (BHANDARKAR; BRUNNER; KALE, 2000)

The interaction between objects and load balancing framework occurs through Object Managers (see Figure 3.3). Object Managers are responsible for creation, destruction and migration of objects. They also supply the load database coordinator with computational loads and communication information of the objects they manage. The tests with this framework were performed using a version of MPI called ArrayMPI and the Converse runtime system. Periodically, the MPI application transfers control to the load balancer using a special call MPI_Migrate(). This call allows the framework to invoke a load balancing strategy and to remap virtual processors (objects) to physical processors. The authors developed two heuristic strategies for load balancing: (i) greedy and; (ii) based on Metis tool.

The greedy strategy organizes all objects in decreasing order of their computation times. The algorithm repeatedly selects the heaviest unassigned object, and assigns it to the least loaded processor, updating the loads, and readjusting the heap. This strategy does not consider data about communication. The other strategy uses the Metis tool which is mainly used tool for partitioning of large structured or unstructured meshes. The object communication graph that is obtained from the load balancing framework is presented to Metis in order to be partitioned onto the available number of processors. The objective of Metis is to find a reasonable load balance, while minimizing the edge-cut. The edge-cut is defined as the total weight of edges that cross the partitions, which in the case denotes the number of messages across processors.

Krivokapic, Islinger e Kemper (2000) presented the concepts of migration and load balancing of objects in WAN environments. For that, these authors developed the AutoO system. This system considers objects that migrate autonomously. Its architecture consists of many sites which execute one or more objects as given in Figure 3.4. Each site has a Site Manager which supports communication between sites and coordinates the processes under its control. The wide area network (WAN) environment is composed by a set of local area networks (LAN). In addition, the authors describe links among the LANs that will inform the communication costs. The composition of wide area networks using local area ones is depicted in Figure 3.5. Two migration policies were designed for this environment: (i) Best-LAN and; (ii) Good-LAN. Both strategies consider only the problem of determining the LAN an object should migrate to. Within a LAN the object chooses the site which most accesses came from.

The Best-LAN strategy considers all possible places (LANs) to migrate an object. It finds the optimal location to an object, but it produces a high computation costs. The second strategy, Good-LAN, considers a good solution but not necessary the optimal one. When an object checks whether it should migrate or not, it considers only a certain number of the latest messages it has received. In order to find the best LAN the object calculates the communication costs these messages would have caused in case the object had been located in other LANs of the system. Accesses from within the same LAN are not taken into account, since the costs they induce are assumed to be much lower than for accesses from outside the LAN. The costs that would have been encountered in the best LAN, i.e., the one with the minimum costs, are denoted as $C_{best-lan}$. The object decides to migrate in case the profit is higher than a certain threshold $t$, as explained in inequality 3.3.

$$C_{curr-lan} - C_{best-lan} \geq t \tag{3.3}$$

The threshold $t$ in Inequality 3.3 presents distinct values depending on the nature of the communication: (i) inside the same LAN or; (ii) with different LANs. Considering this, the value of $t$ is computed taking into account the communication costs inside a

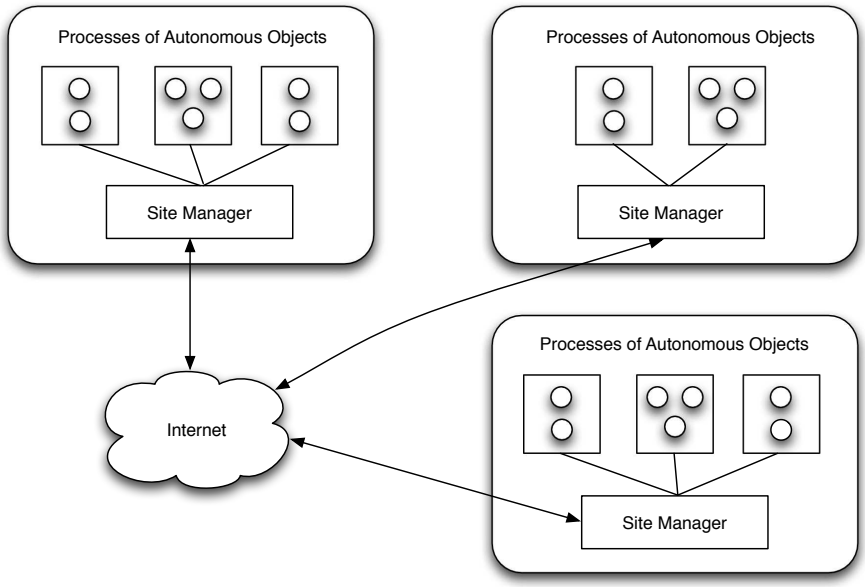


Figure 3.4: Architecture of the system of autonomous objects: AutoO System (KRIVOKAPIC; ISLINGER; KEMPER, 2000)

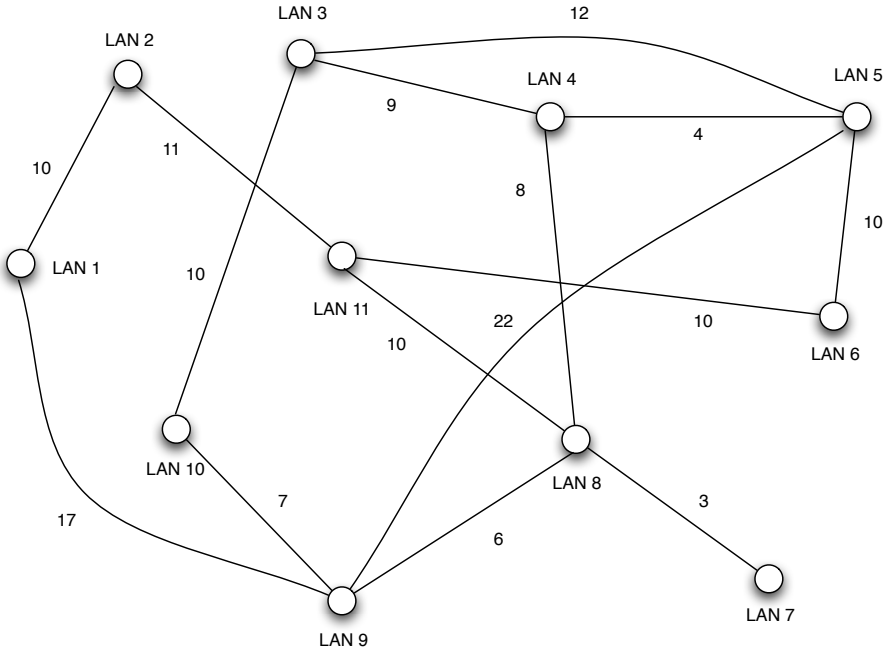


Figure 3.5: Example of a WAN network structure composed by multiple LAN networks (KRIVOKAPIC; ISLINGER; KEMPER, 2000)

LAN ($c_{local}$) and between LANs ($c_{global}$). $t$ is the result of the multiplication of $c$ (local or global) by the number of considered messages. The Algorithm 2 shows the functioning of the strategy Best-LAN. Considering the notations, $O_i$ means an object located initially on machine $M_i$ and LAN $L_i$.

The algorithm for Best-LAN firstly verifies the communication costs for a specific object $O_i$ among all LANs except that it belongs currently. With the best LAN, the algorithm analyses a migration between different LANs based on $c_{global}$. If the inequality for this migration is false, the migration within the same LAN is tested. The strategy

Good-LAN takes profit from the hierarchy of set of LANs. These sets are based on the distance among the networks. In addition, a set can be joined with other ones creating clusters. The network distance between two clusters is given by average message transfer time between them which is determined as the average of the distances between each of the sub-clusters they comprise.

Dynamic load balancing on dedicated heterogeneous system is presented by Galindo et al. (2008). They present three directives for offering load balancing. Two of them are used to initiate and finalize the library and the remaining is employed to perform the load balancing. This last function is called ULL_MPI_calibrate(). It is a collective function that must be invoked by all processes. The authors explain that although a large number of balancing algorithms can be found in the literature, they opted for a simple and efficient strategy that yielded satisfactory results. All processors perform the same balancing operations as follows. Firstly, the time required by each processor to carry out the computation in each iteration has to be given to the algorithm. A collective operation is performed to share these times and other data among all processors. After this operation, each processor presents:

- $T[]$ is vector where each processor gathers all the times ($T_i$);
- *size_problem* is the size of the problem to be computed in parallel;
- *counts*[] is a vector that holds the sizes of the tasks to be computed on each processor.

The next step of the algorithm is to verify that the threshold is not being exceeded. If (Max($T[]$) - Min($T[]$)) > Threshold, then the load balancing should be called. The relative

**Algorithm 2** Algorithm Migration Best-LAN

1: $C_{curr-lan} \leftarrow$ calculateCommCosts($L_i$)
2: best-lan $\leftarrow L_i$
3: $C_{best-lan} \leftarrow C_{curr-lan}$
4: **for** j=0, LAN $L_j \neq L_i$ **do**
5: costs $\leftarrow$ calculateCommCosts($L_j$)
6: **if** costs < $C_{best-lan}$ **then**
7: $C_{best-lan} \leftarrow$ costs
8: best-lan $\leftarrow L_j$
9: **end if**
10: **end for**
11: **if** $C_{curr-lan} - C_{best-lan} > c_{global}$ . number_of_messages **then**
12: Return id of the machine in best-LAN with most access
13: **end if**
14: **if** $L_i$ = best-lan **then**
15: best-machine $\leftarrow$ machine within $L_i$ where most access came from
16: $C_{best-machine} \leftarrow$ calculateCommCosts(best-machine)
17: $C_{M_i} \leftarrow$ calculateCommCosts($M_i$)
18: **if** $C_{M_i} - C_{best-machine} > c_{local}$ . number_of_messages **then**
19: Return the id of the best machine
20: **end if**
21: **end if**
22: Return the id of $M_i$

power $RP$[] is calculated for each processor and corresponds to the relationship between the time $T$[i] invested in performing the computation for a size $counts$[i]. SRP is the sum of all $RP$[i] and is presented in Equation 3.5.

$$RP[i] = \frac{counts[i]}{t[i]} \tag{3.4}$$

$$SRP = \sum_{i=0}^{Numproc-1} RP[i] \tag{3.5}$$

$$counts[i] = size\_problem * \frac{RP[i]}{SRP} \tag{3.6}$$

Finally, the sizes of the new counts are calculated for each processor following Equation 3.6. Each processor fits the size of the task allocated according to its own computational capacity. The system could be extended to run on heterogeneous non dedicated systems and on systems with dynamic load. For that purpose, the array $T[]$ must be fed not only with the execution times but with the loading factor on each processor.

Yu and Shi (2007) proposed an adaptive rescheduling strategy for grid workflow applications. They propose a system design which adapts the Planner to dynamic grid environments via collaboration with the Executor, as shown in Figure 3.6. The system consists of two main components:

- Planner - The Planner has a collective set of subcomponents, including Scheduler, Performance History Repository and Predictor. For each workflow application represented as a DAG, the Planner instantiates a Scheduler instance. Based on the performance history and resource availability, the Scheduler inquires the Predictor to estimate the communication and computation costs with the given resource set. It then decides on resource mapping, with the goal of achieving optimal performance for entire workflow and submits the schedule to the Executor.

- Executor. The Executor is an enactment environment for workflow applications. It can be further decomposed into Execution Manager, Resource Manager and Performance Monitor. The Executor supports advance reservation of resources. The Execution Manager receives the DAG and executes it as scheduled. If the arriving schedule is a result of rescheduling, it revokes resource reservation for replaced schedule before making new reservations.

For a given DAG and a set of currently available resources, the Planner makes the initial resource mapping as any other traditional static approaches do. The primary difference is that this approach requires the Planner to listen for and adapt to the significant events in the execution phase, such as:

- Resource Pool Change - If a new resource is discovered after the current plan is made, rescheduling may reduce the makespan of a DAG by considering the resource addition. When resource fails, fault tolerant mechanism is triggered and it is taken care of by Execution Manager. However, if the failure is predictable, rescheduling can minimize the failure impact on overall performance.

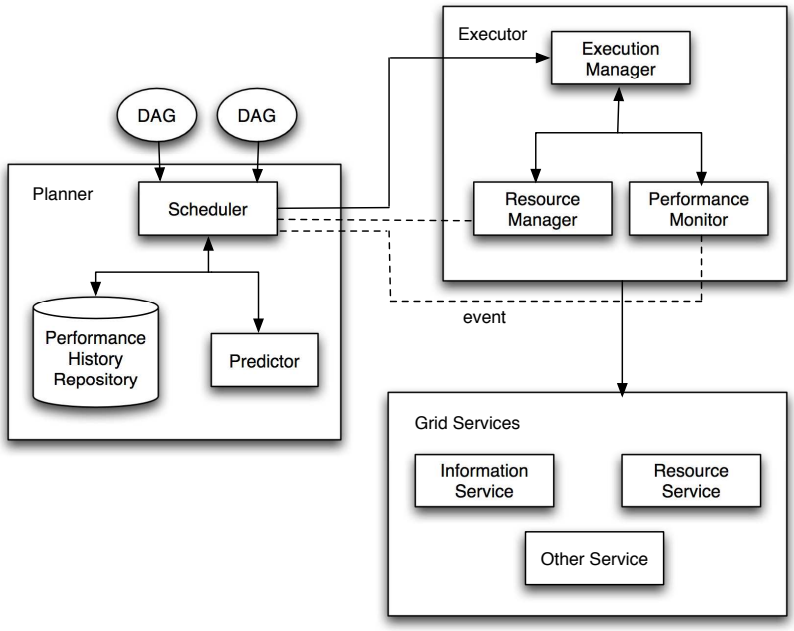


Figure 3.6: The vision of the Planer and Executor components in the rescheduling framework proposed by Yu and Shi (2007)

- Resource Performance Variance - The performance estimation accuracy is largely dependent on history data. An inaccurate estimation leads to a bad schedule. If the runtime Performance Monitor can notify the Planner of any significant performance variance, the Planner will evaluate its impact and reschedule if necessary. In the meantime, the Performance History Repository is updated to improve the estimation accuracy in the subsequent planning.

The Planner reacts to an event by evaluating if makespan can be reduced by rescheduling. For example, if a new resource becomes available, the Planner will evaluate if a new schedule with the extra resource in consideration can produce smaller makespan. If so, the Planner will replace the current scheduling with a new one and will submit it to the Executor.

Utrera et al (UTRERA; CORBALAN; LABARTA, 2005) developed dynamic load balancing in MPI jobs. They propose a mechanism which decides dynamically, without any previous knowledge of the application, whether to apply Local or Global process queues to it by measuring its load balancing degree. One important point on this work is the observation of the authors. Usually the applications at the beginning of execution have an unbalanced behavior, because processes are created and data is distributed among them. After that, they normally perform a global synchronization function. So at this point all the jobs behave as unbalanced ones, after that they start doing regular work. As a matter of fact, the coefficient of variation (CV) of the number of context switches detected on processes during the first period is higher than the rest of the execution owing to the chaotic behavior. This coefficient is used in rescheduling algorithms.

Multi-cluster load balancing based on processes migration is discussed on (WANG et al., 2007). Following the authors, it is a complicated problem to determine the optimal location for the job to be migrated, since available resources are usually heterogeneous and even not measured in the same units. Then, they tried to reconcile these differences by standardizing the resource measurements and adopt a method based on economic princi-

ples and competitive analysis. The target node for migration is determined by computing the opportunity cost for each of the nodes in the local load vector, which is a concept from the field of economies research.

The key idea is to convert the total usage of several heterogeneous resources, such as memory and CPU, into a single homogeneous cost. Jobs are then assigned to the machine where they have the lowest cost, just like in a market oriented economy. A simple way is to compute the marginal cost of a candidate node. Namely, the amount of the sum of relative CPU usage and memory usage would increase if the process was migrated to that node. The goal is to find a node with minimal marginal cost and select it as destination. The dynamic CPU load-balancing algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from high loaded to less loaded nodes.

Du et al. (2004) present a highly configurable and extensible rule-based mechanism for policy making that supports various system conditions in grid environments. The authors designed and implemented a runtime system on top of HPCM and MPI-2, providing resource registration, resource monitoring, process registration and soft-state management to support dynamic rescheduling of MPI tasks. The system has a rule-based decision-making component that coordinates with other components of HPCM as a commander, and invokes the migration when it reaches a migration decision according to a highly configurable and extensible rule-based mechanism.

The model consists of system state monitoring entities, commander entities and a process registration and decision-making entity. Figure 3.7 shows the system model. A monitor and a commander entity reside on each host, including candidate destination hosts. There is also a central or hierarchical registry/scheduler, which can reside on any host with or without other entities. As shown in Figure 3.7, the monitor registers the host static information to the registry/scheduler and periodically gathers and updates the system status to it. The monitor determines the status of its local system resources as free, busy, overloaded or unavailable. It then reports its system status and other information to the registry/scheduler. The registry/scheduler analyzes the data from monitors and makes a decision regarding which process to migrate and where to migrate it.

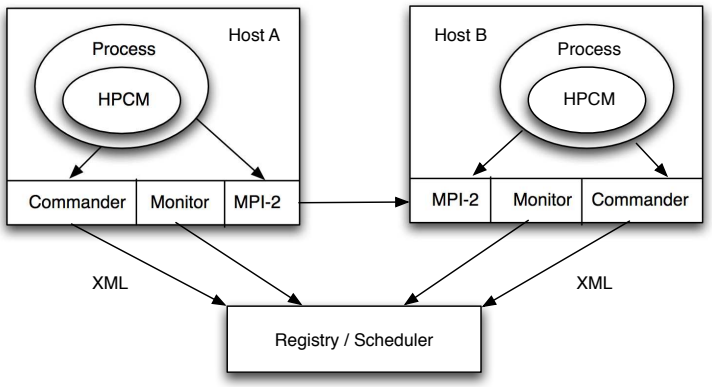


Figure 3.7: The scheduling system model following Du at al. (2004)

The monitors collect various types of information, such as processor utilization and load, memory state, disk usage and communication data (latency and bandwidth). Based on such information, the monitor determines the system status according to a rule-based mechanism. This decision is made locally and specifically according to the rules defined

for a specific local system. Besides data collected at runtime, the model works with detailed application information. Parameters and resource requirements are encapsulated in an application schema with XML format. The application schema contains some information such as: (i) application characteristics, which include data, communication, or computing intensive; (ii) estimated communication data size; (iii) resources requirement and; (iv) estimated execution time on workstation with certain computing power.

Continuing the previous work, Du et al. (2007) analyzed the processes migration costs in shared and distributed environments. A process's state is represented as $S = <App, P, M, IO, Comm>$. The parameters comprise the execution state $P$, the memory state $M$, the I/O state $IO$ and the communication state $Comm$. $f(S)$ is a function that means the size of $S$. In HPCM, data collection, transmission and restoration phases overlap with each other. Lastly, the migration cost can be represented by Equation 3.7.

$$c = \alpha_0 + \mu f(S) \tag{3.7}$$

$\alpha_0$ is a small migration overhead that is application specific. If the application has a large amount of referenced or dynamic allocated memory blocks, $\alpha_0$ is bigger. $\mu$ is called migration processing rate and it is represented as seconds per byte. $\mu$ is proportional to the reciprocal of the current available bandwidth, $b_{ji}$, between the source and destination node, that is $\mu = \frac{\alpha_1}{b_{ji}}$ . $\alpha_1$ is an application dependent constant that reflects the overlapping factor among the migration phases denoted above.

The authors developed a dynamic task scheduling system to reallocate applications at runtime dynamically. To choose a machine as the destination machine, they calculate the expected application execution time after migration and the cost of migration. Let $m_j$ denote the source machine and $m_i$ is the migration destination machine. Let $w'_j$ denote the unfinished workload on $m_j$. The completion time of the unfinished workload $w'_j$, $T_{ji}$, is calculated following Equation 3.8. $T(S'_i)$ represents the execution time of the application with unfinished workload on machine $m_i$.

$$T_{ji} = c_{ji} + T(S'_i) \tag{3.8}$$

After receiving a triggering signal, the scheduling algorithm lists a set of idle machines that are lightly loaded over an observed time period. For each machine, it is computed the migration costs (Equation 3.7) and the mean of the remote task execution time $T(S'_i)$. If the time with migration is lower than the actual scheduling, the migration is performed.

Huang et al. (HUANG et al., 2006) present a performance evaluation of AMPI (Adaptive MPI). The key concept behind AMPI is processor virtualization. Standard MPI programs divide the computation onto $P$ processes, and typical MPI implementations simply execute each process on one of the $P$ processors. In contrast, an AMPI programmer divides the computation into a number $V$ of virtual processors ($VPs$) and AMPI runtime system maps these $VPs$ onto $P$ physical processors. The runtime system has the opportunity of adaptively mapping and remapping the programmer's virtual processors onto the physical machine.

The automatic load balancing idea is described as follows. If some of the physical processors become overloaded, the runtime system can migrate a few of their virtual processors to relatively underloaded physical processors. The runtime system can make such load balancing decision based on automatic instrumentation. AMPI is built on CHARM++ and shares its runtime system and inherits its features. During the execution of an AMPI program, the load balancing framework collects workload information and

object communication pattern on each physical processor in the background. At load balancing time, load balancer uses this information to redistribute the workload, migrating the AMPI threads from overloaded processors to underloaded ones.

Thread migration in AMPI can be done either automatically or with user's help in transferring thread's stack and heap allocated data. Isomalloc stacks and heaps provide a clean way of moving thread's stack and heap data to a new machine by preserving the same address of data across processors. For Isomalloc heaps, user's heap data is given a globally unique virtual address. Then, the heap can be moved to a new machine without changing its address. Isomalloc stacks that AMPI threads run on are allocated from Isomalloc heap. Thus migration is transparent to the user code. Alternatively, users can write their own helper functions to pack and unpack heap data on both processors of a migration. This is useful when application developers wish to reduce the data volume by using application-specific knowledge and/or by packing only variables that are live at the time of migration.

Li and Lan (2005) proposed a migration scheme called DistPM which combines a novel hierarchical approach with real-time performance evaluation. The migration scheme was developed to deal with the heterogeneous and dynamic features of distributed systems. In particular, a heuristic method based on a linear programming algorithm was developed in order to reduce the overhead entailed in migrating workload over shared networks across heterogeneous distributed platforms. One important idea behind the work of Li and Lan is the concept of groups. A group is defined as a set of homogeneous processors connected with dedicated system area networks. A group can be a shared-memory parallel computer, a distributed-memory parallel computer or a cluster of Linux PCs. Each group has two special processes: (i) group coordinators and; (ii) migration gateways.

Each group elects a process as the group coordinator, and a global coordinator is elected from the group coordinators. A coordinator is in charge of collecting group load information, monitoring and predicting both system and application status regarding this group. During each load balancing step, each group coordinator gathers its current load information of the group and reports the information to the global coordinator. Then, this last entity makes a data partition decision with the global knowledge. The principle behind the two-layer coordinators is to maintain the advantages of global knowledge and scalability in the same time while reducing the number of messages among remote processes. In each group, multiple processes are designated as the migration gateways. They are responsible for moving excess workload from overloaded groups to underloaded groups along migration channel. During data migration step, each gateway combine multiple intra-cluster messages into one single message and then sends it to a corresponding gateway in a remote group. The goal here is to minimize the occurrence of remote communications across clusters with high latency and fully utilize the high bandwidth provided in the networks.

Chen, Wand and Lau treat with processes reassignment with reduced migration cost in grid load rebalancing (CHEN; WANG; LAU, 2008). They propose a heuristic that is based on local search and several optimizing techniques which include the guided local search strategy and the multi-level local search. The searching integrates both the changes of workload and the migration cost introduced by a process movement into the movement selection. Furthermore, the searching enables a good tradeoff between low-cost movements and the improving balance level. The algorithm focuses the search in the solution space surrounding the initial assignment, which has a strong likelihood of containing the

desired low-cost solution. By a guided local search technique, the algorithm is able to generate diverse searching trajectories leading to different candidate reassignments, thus increasing the chance to find a reassignment with lower cost. A multi-level local search is proposed, where different restrictions on acceptable migration costs are defined to control the sub-steps of each local search step. The migration cost is a function of two practical parameters: process size and communication bandwidth.

Local search starts with an initial assignment, and searches for a better assignment through a sequence of small steps. A small step refers to one or two process movements between a pair of machines. The authors use 1-move to denote moving a process from one machine to another and 1-swap to denote the exchange of two processes between two machines. Instead of considering movements between all machine pairs, Chen, Wand and Lau permit only movements between the machine with the maximum load level and another machine.

Batista et al. (2008) work with self-adjustment of resource allocation for grid applications. Figure 3.8 presents the self-adjustment idea proposed by Batista et al.. In the step 3 we have the resources monitoring in order to detect any variation in availability of resources, either decreasing or increasing. Step 5 derives a new DAG representing current computational and data transfer demands and produce a schedule for these tasks. Step 7 compares the cost of the solution derived in Step 5 with the cost of the current solution. The cost of the solution derived in Step 5 should include the cost of migrating tasks. A task is only worth moving if a reduction in execution time compensates for the cost. Finally, Step 8 migrates tasks to the designated hosts on the basis of the most recent schedule. Rescheduling decisions consider resource availability and current execution status, as well as the initial scheduling. The proposed procedure does not deal with uncertainties in task demands. Moreover, the programmer must indicate checkpoints inside tasks for their rescheduling and migration.

## 3.3 Grid Computing

This section will explain some works in the field of grid computing. The idea is to start with general ideas and algorithms about scheduling and rescheduling, as well as load balancing and migration in grid computing. After that, we will discuss some systems that present these facilities.

### Self-Organization based on Agents

Cao (2004) presented a self-organizing strategy based on agents for grid load balancing. In this work, an agent-based self-organization strategy is proposed to perform a complementary load balancing for batch jobs with no explicit execution deadlines. In particular, an ant-like self-organizing mechanism is introduced and proved to be powerful to achieve overall grid load balancing through a collection of very simple local iterations. The iterative algorithm proceeds as follows.

1. An ant wanders from one agent to another randomly and tries to remember the identity of an agent that is most overloaded;

2. After a certain number of steps ($m$), the ant changes the mode to search a most underloaded agent, though still wandering randomly;

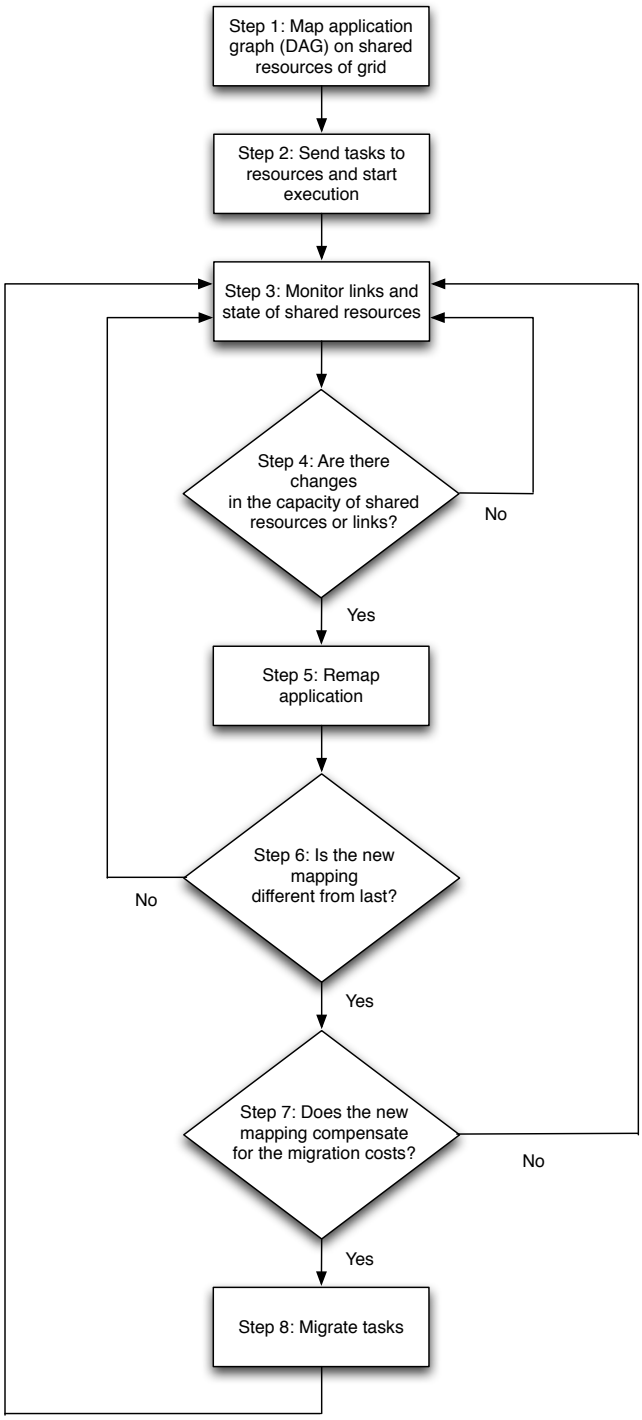


Figure 3.8: Self-Adjustment idea presented by Batista et al. (2008)

3. After the same *m* steps, the ant stops for one step to suggest the current two remembered agents (considered to be most overloaded and underloaded, respectively) to balance their workload;

4. After load balancing is performed, the ant is initialized again and starts a new loop from step 1.

The resource manager works with *n* and *m* parameters, where *n* represents the amount of ants and *m* the number of steps that each one performs at each iteration. The load

balancing speed can be improved increasing the number of ants. If a large amount of them are active simultaneously, the load balancing is done more quickly. On the other hand, this causes a larger costs due to a higher communication among the agents. Besides this issue, if an interaction includes a low number of steps $m$, one ant will start the load balancing with high frequency.

Salehi e Deldari (2006) affirm that the algorithm proposed by Cao (2004) is inefficient since each ant wanders always $2m+1$ steps at each iteration in order to execute the load balancing. Thus, they proposed an echo system where the ants are created according to the demand. They are created when the system is drastically unbalanced and commit suicide when detecting equilibrium in the environment. Each ant in this new algorithm wanders $m$ steps (this value is determined adaptatively) instead $2m+1$. Besides this, in their solution, at the end of $m$ steps, $k$ overloaded nodes are balanced with underloaded ones, instead just one in the first approach.

**Utility Computing**

Utility computing is becoming a popular way of exploiting the potential of computational grids. In utility computing, users are provided with computational power in a transparent manner similar to the way in which electrical utilities supply power to their customers. Aiming to take full advantage of utility computing, an application needs to be mobile; that is, it needs to be able to migrate between heterogeneous computing platforms while it is executing. Furthermore, it needs to be able to adapt to the computing resources at each site, such as the number of available physical processors.

Fernandes, Pingali and Stodghill (FERNANDES; PINGALI; STODGHILL, 2006) describe the PC3 system, which converts C/MPI codes into mobile programs almost transparently. Because it is based on portable application-level checkpointing, it enables the state of running applications to be saved so that the application can be restarted on different architectures, operating systems and MPI implementations. Moreover, the number of processors on these machines can be different. PC3 system is based on three key ideas, listed below.

- Over-decomposition - The authors decompose the original MPI application into a large number of virtual processes. This allows us to execute the application on different numbers of processors and obtain good load-balance by appropriately assigning virtual processes to physical processors.

- Application-level checkpointing - The authors use a pre-compiler to instrument a C program so that it can save and restore its own state without relying on operating system or hardware support. Unlike system-level checkpointing systems like Condor that are tied to particular architectures and operating systems, application-level checkpointing provides an approach for making programs self-checkpointing and self-restarting.

- Coordination layer - To save the state of a parallel program while it is executing, it is necessary to coordinate the state-saving by the different processes. If the program is written in a bulk-synchronous manner, the state of the computation can be saved at global barriers. However, following the authors, some programs such as many mesh generators do not have global barriers. To checkpoint such programs, PC3 system uses a coordination protocol implemented by a thin software layer, which intercepts all calls made by the program to the MPI library.

The system uses type information to save the state of the MPI processes in a portable manner, and uses either barrier or non-blocking protocols for coordinating the MPI state. The experimental results indicate that the overheads of checkpointing are less than 2% on average for the benchmarks used in the paper.

**GridWay Resource Manager**

The GridWay framework is a Globus compatible environment, which simplifies the user interfacing with the grid (MORENO-VOZMEDIANO; ALONSO-CONDE, 2005). It provides resource brokering mechanisms for the efficient execution of jobs on the grid, with dynamic adaptation to changing conditions. The main component of GridWay framework is the Dispatch Manager, which is responsible for job scheduling. It is also responsible for allocating a new resource for the job in case of migration (rescheduling). The migration of a job or a job subtask can be initiated for the following reasons: (i) a forced migration requested by the user; (ii) a failure of the target host and; (iii) the discovery of a new better resource, which maximizes the optimization criterion selected for that job.

GridWay resource broker supports scheduling and migration under different user optimization criteria (time optimization and cost optimization). The goal of the time optimization criterion is to minimize the completion time for the job. To meet this optimization criterion, the Dispatch Manager must select those computing resources being able to complete the job as faster as possible, considering both the execution and the file transfer times. Thus, the mechanism returns a prioritized list of resources, ordered by a rank function that must comprise both the performance of every computing resource and the file transfer delay. The time optimization rank function used in this implementation can be seen in Equation 3.9.

$$TR(r) = PF(r) \,.\, (1 \,-\, DF(r)) \tag{3.9}$$

In Equation 3.9, $TR(r)$ means the time optimization rank function for resource $r$ and $PF(r)$ is a performance factor for resource $r$. In addition, $DF(r)$ is a file transfer delay factor for resource $r$. $PF(r)$ is computed as the product of the peak performance (MHz) of the target machine and the average load of the CPU in the last 15 minutes. $DF(r)$ is a weighted value in the range [0-1], which is computed as a function of the time elapsed in submitting a simple job to the candidate resource. To avoid worthless migrations, the rank function of the new discovered resource must be at least 20% higher than the rank function of the current resource. Otherwise the migration is rejected.

The goal of the cost optimization criterion is to minimize the CPU cost consumed by the job. In this model, the authors only considered the computation expense (i.e. the CPU cost). To minimize the CPU cost, the resource selector must know the rate of every available resource, which is usually given in the form of price (Grid $) per second of CPU consumed. Once the resource selector has got the price of all the resources, it returns an ordered list using the rank function described in Equation 3.10. As in the first case, the migrations only occur if a profit greater then 20% is reached with the new resource.

$$CR(r) = \frac{PF(r)}{Price(r)} \tag{3.10}$$

Montero, Huedo and Llorente developed a grid resource selection for opportunistic job migration (MONTERO; HUEDO; LLORENTE, 2003). They put this functionality on GridWay resource broker. In (MONTERO; HUEDO; LLORENTE, 2003), the Dispatch

Manager wakes up at each discovery interval and tries to find a better host for each job by activating the resource selection process. In order to evaluate the benefits of job migration from the current execution host ($h_{n-1}$) to each candidate host ($h_n$), Montero, Huedo and Llorente define the migration gain ($G_m$) following Equation 3.11.

$$G_m = \frac{T_{exe}(h_{n-1}, t_{n-1}) - T_{exe}(h_n, t_n)}{T_{exe}(h_{n-1}, t_{n-1})} \tag{3.11}$$

$T_{exe}(h_{n-1}, t_{n-1})$ is the estimated execution time on current host when the job was submitted to that host, and $T_{exe}(h_n, t_n)$ is the estimated execution time when the application is migrated to the new candidate host. The migration is granted only if the migration gain is greater than an user-defined threshold, otherwise it is rejected. Note that although the migration threshold is fixed for a given job, the migration gain is dynamically computed to take into account the dynamic data transfer overhead, the dynamic host performance and the application progress. The migration gain was fixed to 10% in the experiments presented in (MONTERO; HUEDO; LLORENTE, 2003).

### Hierarchical Scheduling in EasyGrid

The EasyGrid middleware is a hierarchically distributed Application Management System embedded into MPI applications to facilitate their efficient execution in computational grids (BOERES et al., 2005). The overhead of employing a distinct AMS (Application Management System) to make each application system aware does however bring at least two benefits. Firstly, the scheduling policies adopted can be tailored to the specific needs of each application leading to improved performance. Secondly, distributing the management effort amongst the applications themselves makes grid management more scalable. Figure 3.9 depicts the AMS-based MPI system. The AMS is embedded automatically into an user's parallel MPI application without modifications to the original code.

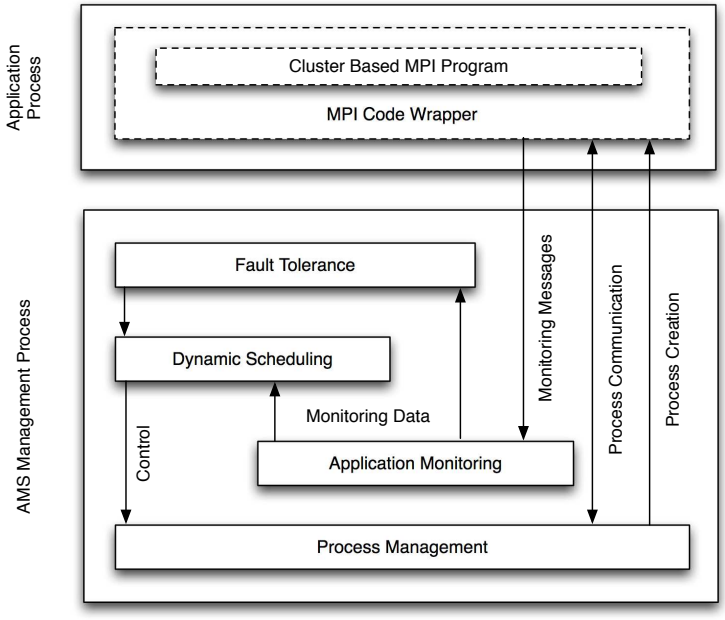


Figure 3.9: The subsumption architecture of an MPI AMS management process (BOERES et al., 2005)

Each EasyGrid AMS is a three level hierarchical management system composed of the following entities: (i) a single Global Manager (GM), at the top level, which supervises the sites in the grid where the application is running (or could execute); (ii) at each of these sites, a Site Manager (SM) is responsible for the allocation of the application processes to the resources available at the site and; (iii) Host Manager (HM) which resides on each resource and takes the responsibility for scheduling, creation and execution of the application processes associated with that respective host. Figure 3.10 illustrates an example of an hierarchical grid structure with these three entities.

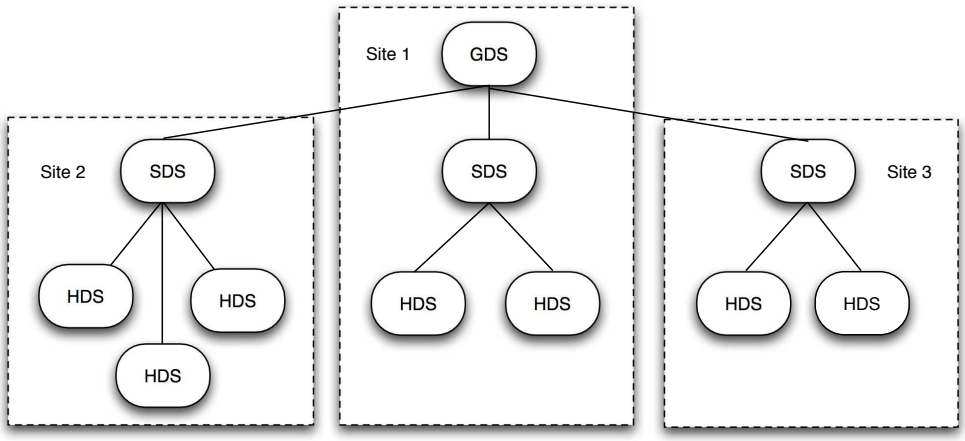


Figure 3.10: AMS hierarchical schedulers (BOERES et al., 2005)

The EasyGrid AMS can perform static and dynamic scheduling. In the first case, the parallel application is represented by directed acyclic graphs (DAGs) and the relevant characteristics of the target system (e.g. computing power, communication costs) are captured by an architectural model. In a DAG $G = (V, E, \varepsilon, \omega)$: the set of vertices, $V$ , represent tasks; $E$, the precedence relation among them; $\varepsilon(v)$ is the amount of work associated to task $v \in V$ ; and $\omega(u,v)$ is the weight associated to the edge $(u,v) \in E$, representing the amount of data units transmitted from task $u$ to $v$. In dynamic scheduling approach, EasyGrid AMS only reschedules processes which have yet to be created. The dynamic schedulers are associated with each of the management processes distributed in the three levels of the AMS hierarchy. As shown in Figure 3.10, the Global Dynamic Scheduler (GDS), at the top level, is executed in the GM, while each SM has a Site Dynamic Scheduler (SDS) and HM, a Host Dynamic Scheduler (HDS) respectively.

Collectively, the dynamic schedulers repeatedly estimate the remaining execution time of the application workload on each resource and, consequently, the makespan of the application. They verify if the allocation needs to be adjusted and, if necessary, activate the rescheduling mechanism. Since existing processes are not migrated, the AMS dynamic scheduling strategy is based on anticipation. Processes are rescheduled before any imbalance affects the performance. The interval between the scheduling events depends on the variation in the performance offered by grid resources and the threshold values.

**Self-Adaptivity on GrADS**

Vadhiyar and Dongarra discuss the design and implementation of a software system that dynamically adjusts the parallelism of applications executing on computational grids in accordance with the changing load characteristics of the underlying resources (VADHIYAR; DONGARRA, 2005a). The migration framework implements tightly coupled

policies for both suspension and migration of executing applications. The suspension and migration policies consider both the load changes on systems as well as the remaining execution times of the applications. In their framework, the migration of applications depends on:

- The amount of increase or decrease in loads on the resources;
- The point during the application execution lifetime when load is introduced into the system;
- The performance benefits that can be obtained for the application due to migration.

The migration framework is primarily intended for rescheduling long running applications. The migration is dependent on the ability to predict the remaining execution times of the applications, which in turn is dependent on the presence of execution models that predict the total execution cost of the applications. The ability to migrate applications in the GrADS system is implemented by adding a component called Rescheduler to the GrADS architecture. Migrator, Contract Monitor and Rescheduler are the three main components of GrADS. Migrator is responsible to effectivate the migration, while the Contract Monitor monitors the applications' progress. Rescheduler decides when to migrate an application and the set of resources.

The authors implemented an user-level checkpoint library called SRS (Stop Restart Software). This library is implemented on top of MPI and hence can be used only with this kind of parallel programs. The rescheduler is a daemon that operates in two modes: (i) migration on request and; (ii) opportunistic migration. When the Contract Monitor detects intolerable performance loss for an application, it contacts the Rescheduler requesting it to migrate the application. This is called migration on request. In other cases when Contract Monitor does not contact the Rescheduler for migration, the Rescheduler periodically queries the GrADS repository for recently completed applications. The rescheduler determines if performance benefits can be obtained for an executing application by migrating it to free resources. This is called opportunistic rescheduling.

GrADS project minimizes the overall job completion time, also know as makespan (BERMAN et al., 2005). Based on the total percentage completion time for the application and the total predicted execution time for the application with the new schedule, the rescheduler calculates the remaining execution time (*ret_new*) of the application. The rescheduler also calculates *ret_current*, the remaining execution time if the application continues executing on the original set of machines. Considering this, the rescheduler calculates the rescheduler gain as reveals Equation 3.12.

$$rescheduler\_gain = \frac{(ret\_currrent - (ret\_new + 900))}{ret\_current} \tag{3.12}$$

The number 900 in the numerator of the fraction is the worst case time in seconds to reschedule the application. If the rescheduling gain is greater than 30%, the rescheduler migrates the application. Berman et al. (BERMAN et al., 2005) presented new scheduling and rescheduling algorithms for GrADS. Firstly, the rank of the resource $r_j$ is calculated by using a weighted sum of the expected execution time on the resource and the expected cost of data movement for the component $c_i$. Equation 3.13 explains how the rank can be calculated.

$$rank(c_i, r_j) = w1.eCost(c_i, r_j) + w2.dCost(c_i, r_j) \tag{3.13}$$

The expected execution time is expressed by *eCost*, while *dCost* is the cost of data movement. A performance matrix is constructed in order to evaluate the ranks of the processes. Each element of the matrix $p_{i,j}$ denotes the rank value of executing the $i^{th}$ component on the $j^{th}$ resource. This matrix is used by the scheduling heuristics to obtain a mapping of components onto resources. Such a heuristic approach is necessary since the mapping problem is NP-complete. The authors apply three heuristics to obtain three mappings and then select the schedule with the minimum makespan. The heuristics are: (i) min-min; (ii) max-min and; (iii) sufferage.

Normally, a contract violation activates the GrADS rescheduler. The rescheduling process must determine whether rescheduling is profitable, based on the sensor data, estimation of the remaining work in the application, and the cost of moving the application to new resources. If rescheduling appears profitable, the rescheduler computes a new schedule and contacts rescheduling actuators located on each processor. There are two techniques for rescheduling: (i) rescheduling by stop and restart and; (ii) rescheduling by processor swapping. In the first approach, the application is suspended and migrated only when better resources are found for application execution. Although very flexible, the natural stop, migrate and restart approach to rescheduling can be expensive. The process swapping method is an alternative. To enable swapping, the MPI application is launched with more machines than will actually be used for the computation. Some of these machines become part of the computation (the active set) while some do nothing initially (the inactive set). During execution, the Contract Monitor periodically checks the performance of the machines and swaps slower machines in the active set with faster machines in the inactive set.

### InteGrade Middleware

Andrei et al. (2005) developed an implementation of the BSP model for the InteGrade middleware. Its usability is focused on computers sharing located in laboratories, local networks and clusters. InteGrade takes profit from the power of idle computers to execute useful applications. The user may specify execution prerequisites such as software and hardware platforms, minimal requirements of memory, as well as a rule like: select the highest available CPU.

InteGrade presents local and global resource managers, called LRM and GRM respectively. They cooperate in order to treat the resource management. LRM is executed in each cluster node, collecting data about its state, CPU, memory and network usage. LRM managers send these data to GRM periodically. GRM uses them aiming to perform the scheduling inside a cluster. Thus, GRM selects the candidate nodes based on available resources and application requirements. It negotiates with the nodes and applies the reservation for application execution. GRM is also responsible for inter-cluster communication.

## 3.4 Bulk Synchronous Parallel Model

BSP is a model for parallel programming that was planned for applications that execute on homogeneous and dedicated resources. However, we can use it on grid computing. Thus, some researchers attempt to apply the BSP model to the grid environment. Vasil P. Vasilev (2003) proposes a BSPGRID model, which exploits the bulk synchronous paradigm to allow existing algorithms to be easily adapted and used in grids. By modifying the BSPGRID model, Jeremy M. R. and Alexander V. provide the Dy-

namic BSP (MARTIN; TISKIN, 2004), which deals with fault tolerance issue in grid environments. Tong W. et al. present the BSP-G (WEIQIN; JINGBO; LIZHI, 2003), a BSP development library based on the services of Globus Toolkit. Song Jiong et al. put forward a ServiceBSP model, which deals with the QoS issue on grid services (SONG; TONG; ZHI, 2006).

Besides computational grids, the rules of the BSP model is useful in data-grids. Data-grids are becoming increasingly important for sharing large data collections, achievements and resources. Chen and Tong (CHEN; TONG, 2004) had the idea that a superstep in the BSP model should be able to help data-grid access and storage in regular sequence. When services are not isolated from each other in multi-user environment, this should be able to avoid, in the process of data access and storage, the occurrence of four types or phenomena: (i) Lost update; (ii) Dirty read; (iii) Non-repeatable read and; (iv) Phantom. Following the document, we will present some initiatives that are based on BSP model. These initiatives provide a middleware for specific functionalities without changing the BSP model or propose some extension in this model to work with specific situations.

**ServiceBSP Model**

Jiang et al. (JIANG; TONG; ZHAO, 2007; ZHU; TONG; DONG, 2006; SONG; TONG; ZHI, 2006) developed the ServiceBSP model. ServiceBSP model combines parallel computing model BSP and the concept of service. In this context, load balancing is also a demand driven issue. The dynamic load balancing proposed by the authors is based on a load rank of the nodes. Service Scheduling Agent sends the new task to the currently lightest node. In proposed method, the authors access the load of all the nodes relying on dynamic load value. They calculate the value taking such factors into account as CPU, memory resource, number of current tasks, response time, the number of network links, etc. A load value of a node owning only one task can be described by follow Equation 3.14.

$$Vi = M1 * Vcpu + M2 * Vmem + M3 * Vio + M4 * Vtasks + M5 * Vnetwork + M6 * Vresponse \quad (3.14)$$

Vcpu, Vmem, Vio, Vtasks, Vnetwork, Vresponse respectively represent the usage ratio of CPU, the usage of memory, the number of current tasks, the number of network links, response time. The load value of a node owning several tasks is V where V=$\sum_{i=0}^{n} Vi$. Using the current load value V, the authors can calculate a new value when a new task is joining.

**H-BSP: Hierarchical Bulk Synchronous Parallel Model**

The original BSP model does not take into account the locality of the processor and the hardware that is being used to execute the application. In order to complete this gap, Cha and Lee created the model H-BSP (Hierarchical Bulk Synchronous Parallel) (CHA; LEE, 2001). H-BSP adopts BSP as its sub-model and offers an mechanism to take advantage of processor locality and thus enables the development of more efficient algorithms. In addition to the basic BSP principle, H-BSP uses a special mechanism to split the entire system into a number of smaller groups. The system is dynamically split or merged at runtime. Each group behaves as an independent BSP system and they communicate in an asynchronous fashion.

Figure 3.11 illustrates, with an example, how an H-BSP algorithm works in general.

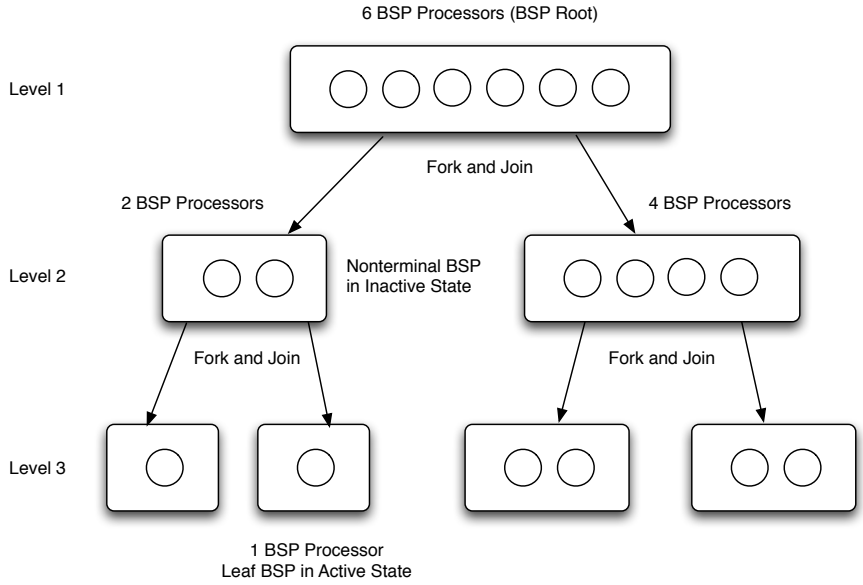


Figure 3.11: Concepts of H-BSP involving three levels of hierarchy (CHA; LEE, 2001)

The level-1 BSP, called a root BSP, initially consists of 6 processors. It then splits into two level-2 groups; 2-processor BSP and 4-processor BSP. The split process continues until no further split is possible in the group; *i.e*. all BSP groups are a single processor group. The dynamically created BSPs run, within each group, in a bulk synchronous fashion but they are independent of each other and run asynchronously. At any time, only the leaf BSP groups are in active state. Their parents, the non-terminal BSP groups, should wait until their children BSPs (leaf BSPs) terminate.

**Dynamic BSP Model**

Martin and Tiskin (2004) present a significant modification to the BSPGRID approach, which will enable to address the heterogeneity issues, as well as fault-tolerance. They created the Dynamic BSP model, which offers a more flexible programming model, with the ability to spawn additional processes within supersteps when required. The essence of DynamicBSP is to use the task-farm model to implement BSP supersteps, where the individual tasks correspond to processes (see Figure 3.12). The authors propose a mechanism to avoid the data bottleneck. Their model consists of: (i) a master processor (task server); (ii) worker processors and; (iii) a data server (which can either be implemented as distributed shared memory or remote/external memory). In each superstep there is a bag of virtual processors to be run on a pool of available physical processors.

The master processor is responsible for task scheduling, memory management, and resource management. At the beginning of each superstep, a virtual processor number is distributed to each physical processor, which then has the responsibility to retrieve local data from the data server, perform the required computations, write back the modified data, and then inform the master processor that it has finished the task. The master processor maintains a queue of pending virtual processors and dynamically assigns them to waiting physical processors. As soon as all the virtual processors have been executed on a particular superstep, the global shared memory is restored to a consistent state and the next superstep commences. If a physical processor fails to complete its task within a reasonable time, then it is considered as dead and its work is reallocated to another physical processor within the same superstep.

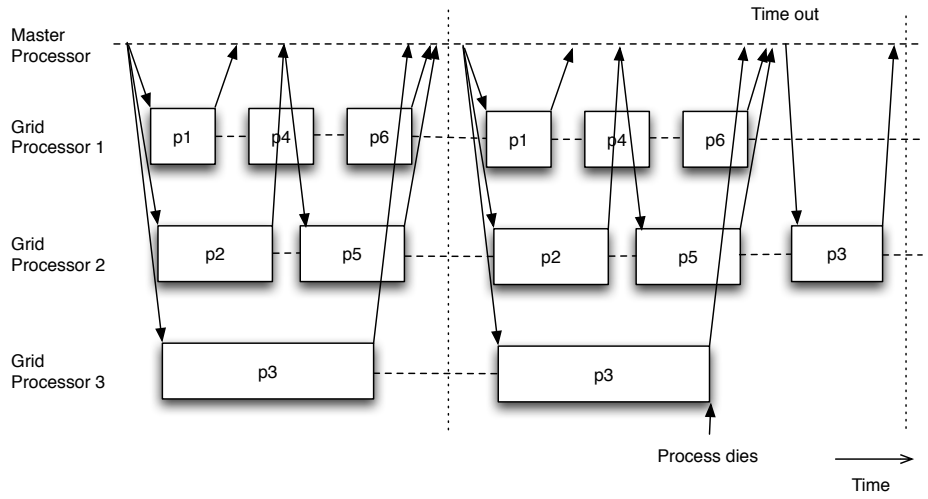


Figure 3.12: Example of DynamicBSP application execution (MARTIN; TISKIN, 2004)

**HBSP: Heterogeneous Bulk Synchronous Parallel Model**

The Heterogeneous Bulk Synchronous Parallel (HBSP) model is a generalization of the BSP model (WILLIAMS; PARSONS, 2000). BSP is inappropriate for heterogeneous parallel systems since it assumes all components have equal computation and communication abilities. HBSP enhances the applicability of BSP by incorporating parameters that reflect the relative speeds of the heterogeneous computing components. An HBSP computer is characterized by the following parameters.

- The number of processor-memory components $p$ labeled $P_0, ..., P_{p-1}$;
- The gap $g_j$ $j \in [0...p-1]$, a bandwidth indicator that reflects the speed with which processor $j$ can inject packets into the networks;
- The latency $L$, which is the minimum duration of a superstep, and which reflects the latency to send a packet through the network as well as the overhead to perform a barrier synchronization;
- Processor parameters $c_j$ for $j \in [0...p-1]$, which indicates the speed of processor $j$ relative to the slowest processor, and;
- The total speed of the heterogeneous configuration $c = \sum_{i=0}^{p-1} c_i$.

For notational convenience, $P_f$ and $P_s$ represent the fastest and the slowest processors, respectively. The communication time and the computation speed of the fastest (slowest) processor are $g_f$ ($g_s$) and $c_f$ ($c_s$), respectively. The authors assume that $c_s$ is normalized to 1. If $c_i = j$, then $P_i$ is $j$ times faster than $P_s$. Execution time of an HBSP computer is determined as follows. Each processor, $P_j$, can perform $w_{i,j}$ units of work in $\frac{w_{i,j}}{c_j}$ time units during superstep $i$. Let $w_i = max(\frac{w_{i,j}}{c_j})$ represent the largest amount of local computation performed by any processor during superstep $i$. Let $h_{i,j}$ be the largest number of packets sent or received by processor $j$ in superstep $i$. Thus, the execution time of superstep $i$ is reached by Equation 3.15

$$S = w_i + max\{g_j.h_{i,j}\} + L \tag{3.15}$$

The overall execution time is the sum of the superstep execution times. The more complex cost does not change the basic programming methodology, which relies on the superstep concept. Furthermore, when $c_j = 1$ and $g_j = g_k$, where $0 \leq j,k \leq p$, HBSP is equivalent to BSP.

**PUBWCL: Paderborn University BSP-based Web Computing Library**

Bonorden developed the PUBWCL library (BONORDEN; GEHWEILER; HEIDE, 2005). The Paderborn University BSP-based Web Computing Library (PUBWCL) is a library for parallel algorithms designed according to the BSP model and intended to utilize the unused computation power on computers distributed over the Internet. Participants willing to donate their unused computation power have to install a PUBWCL client. Whenever an user wants to execute a BSP program, the system chooses a subset of these clients to run the program.

PUBWCL can migrate BSP processes during the barrier synchronization (and additionally at arbitrary points specified by the developer of a BSP program). Furthermore, BSP processes can be restarted on other clients when a PC crashes or disconnects from the Internet unexpectedly. This feature is also used to enable load balancing strategies to abort a BSP process if it does not finish within some deadline, and restart it on a faster client. The author reduces the scheduling problem for a BSP algorithm with $n$ supersteps to $n$ subproblems, namely scheduling within a superstep. Bonorden implemented the following load balancing strategies.

- Algorithm PwoR - Whenever a superstep is completed, all clients are checked whether the execution of the BSP processes on them took more than $r$ times the average execution duration. In this case, the BSP processes are redistributed among the active clients such that the expected execution duration for the next superstep is minimal, using as little migrations as possible.

- Algorithm PwR - Using the load balancing algorithm Parallel Execution with Restarts (PwR), the execution of a superstep is performed in phases. The duration of a phase is $r$ times the running time of the $\lceil s.p^* \rceil$-th fastest of the (remaining) BSP processes, where $p^*$ is the number of processes of the BSP program that have not yet completed the current superstep. At the end of a phase, all incomplete BSP processes are aborted. In the next phase, they are restored on faster clients.

- Algorithm SwoJ - While the two load balancing strategies PwoR and PwR execute all BSP processes in parallel, the load balancing algorithm Sequential Execution without Just-in-Time Assignments (SwoJ) executes only one process of a BSP program per client at a time. The other BSP processes are kept in queues. Like PwR, SwoJ operates in phases. At the end of a phase all uncompleted BSP processes are aborted and reassigned.

- Algorithm SwJ - Like SwoJ, the load balancing algorithm Sequential Execution with Just-in-Time Assignments (SwJ) executes only one process of a BSP program per client at a time and keeps the other processes in queues, too. The main difference, however, is that these queues are being balanced. More precisely, whenever a client has completed the execution of the last BSP process in its queue, a process is migrated to it from the queue of the most overloaded client.

Bonorden also developed an extension of the PUB library to support processes migration (BONORDEN, 2007). He implemented three strategies for load balancing. The first one is a centralized method in which one node gathers the load of all nodes and makes all migration decisions. The others employ distributed method without global knowledge. All load balancing strategies are explained in detailed below.

- The global strategy - All nodes send information about their CPU-Power and their actual load to one master node. This node calculates the least ($P_{min}$) and the most ($P_{max}$) loaded node and migrates one virtual processor from $P_{max}$ to $P_{min}$ if necessary.

- Simple distributed strategy - Every node asks a constant number $c$ of randomly chosen other nodes for their load. If the minimal load of all the $c$ nodes is smaller than the own load minus a constant $d \geq 1$, one process is migrated to the least loaded node.

- Global prediction strategy - Each node has an array of the loads of all other nodes but these entries are not up to date all the time. Each node sends its load periodically to $k$ uniformly at random chosen nodes in the network.

All these strategies use the one minute load average methodology. In addition, they all take into consideration neither the communication among the processes, nor the migration costs.

## 3.5 Summary

Considering the spectrum of related work presented in this chapter, we can emphasize some positive features of the studied systems and algorithms. They are presented below:

- Hierarchical Scheduling - The notion of the hierarchical and cooperative scheduling is a pertinent idea to divide the scheduling work (LI; LAN, 2005; CHA; LEE, 2001). We observed that the use of coordinators is being employed in this context (CHEUNG; JACOBSEN, 2006; BOERES et al., 2005; KRIVOKAPIC; ISLINGER; KEMPER, 2000). Each coordinator can interact with local nodes under its jurisdiction and passes scheduling messages to other coordinators in the same level or above it in the hierarchy.

- Multiple metrics to compute the load - The combination of two or more metrics such as computation and communication to compute the migration candidates is important on dynamic and heterogeneous environments as the grids can be modeled. This combination provides a more precise on the fly decision making to achieve performance during application runtime (HEISS; SCHMITZ, 1995; BHANDARKAR; BRUNNER; KALE, 2000; KONDO et al., 2002; HARCHOL-BALTER; DOWNEY, 1997; YOUNG et al., 2003).

- Automatic rescheduling - The act to perform the load rebalancing automatically through migration is pertinent since direct modifications in application code are not needed. Besides this topic, other is the fact that any interactions from the programmer/user in the environment are not necessary during application runtime.

Taking into account the scope of proposals that follow this idea, we can cite (KRIVOKAPIC; ISLINGER; KEMPER, 2000; THAIN; TANNENBAUM; LIVNY, 2005; VADHIYAR; DONGARRA, 2005a; CAO, 2004; HUANG et al., 2006; UTRERA; CORBALAN; LABARTA, 2005).

- High Throughput Computing - This area explores computing cycles of idle machines. Normally, the projects here use machines that are located around the world through Internet connection (THAIN; TANNENBAUM; LIVNY, 2005; KONDO et al., 2002; BONORDEN; GEHWEILER; HEIDE, 2005; GOLDCHLEGER et al., 2005).

- Performance prediction - The application execution time prediction is explored in the following publications (HARCHOL-BALTER; DOWNEY, 1997; VADHIYAR; DONGARRA, 2005a; SPOONER et al., 2003). In addition, the use of metadata that save previous processing results and network data is explained in ICENI Middleware (YOUNG et al., 2003). In general, the use of data of previous executions in order to obtain better application performance is discussed in (YU; SHI, 2007; MONTERO; HUEDO; LLORENTE, 2003; VADHIYAR; DONGARRA, 2005a; DU et al., 2004; LI; LAN, 2005). Besides this, Krivokapic, Islinger and Kemper (2000) employ the communication metric in order to select the target destination of migration tasks. Their idea is to find out the best location taking into account that future accesses will be similar to those occurred in the recent past. Goldchleger et al. (GOLDCHLEGER et al., 2004) developed usage patterns to perform task scheduling. In local level, LUPA (Local Usage Pattern Analyzer) modules obtain data about how users explore resources like CPU, memory and disc. These data are captured as temporal series and are used to predict how free or busy a specific resource will be during the day.

- Self-Organizing feature - Self-organizing capability using agents aiming to to provide migration-based load balancing is described by Cao (2004), as well by Salehi and Tahvil (2005). Following this branch of self-organizing, Vadhiyar and Dongarra (VADHIYAR; DONGARRA, 2005a) present the migration facility on GrADS using a performance detector that verifies the application execution and activates the processes migration mechanism automatically.

- Dynamic control of new resources - DynamicBSP (MARTIN; TISKIN, 2004) middleware changes the BSP model in order to provide flexibility and fault-tolerance. This middleware acts following the master-slave paradigm. It turns possible the master to launch slave processes in new resources during application runtime. Each process can create other ones, making the development of divide-and-conquer applications easier.

- Considering the memory data on processes migration viability calculus - LSF scheduling system (LUMB; SMITH, 2004) analyses the amount of process memory to decide about the viability of its migration. Furthermore, we can cite the effort performed by Heiss and Schmitt (HEISS; SCHMITZ, 1995). They present a decentralized and dynamic load balancing model that considers the migration costs to decide the transferring, or not, of one or more tasks.

- Use of cluster and local networks - The union of several clusters as a viable environment for parallel computing was demonstrated in (YOUNG et al., 2003). In

addition, the use of two or more local networks (LANs) to compose a wide area network environment was explored in (KRIVOKAPIC; ISLINGER; KEMPER, 2000; YAGOUBI; MEDEBBER, 2007).

Considering the item that comprises the use of several metrics in order to compose the notion of load, there are works that join the computation and communication parts of applications (BHANDARKAR; BRUNNER; KALE, 2000; CHEUNG; JACOBSEN, 2006; MORENO-VOZMEDIANO; ALONSO-CONDE, 2005; YOUNG et al., 2003). The observation of the costs related to the processes migration, as well as the transferring rate between nodes is a tactic utilized by Harchol-Balter and Downey (HARCHOL-BALTER; DOWNEY, 1997). Kondo et al. (2002) execute the processes migration in client-server environments with the following metrics: (i) CPU; (ii) network (bandwidth between the client and the server) and; (iii) disc. The authors normalize these values to work units and they take the lowest value to decide the load to be sent to a specific client.

Heiss and Schimitz (HEISS; SCHMITZ, 1995) developed a load balancer where the load of each task is represented by a particle. Such work considers the processors load, the communication among the tasks and the amount of data to be migrated. Moreover, such paper considers static information about the behavior of the tasks (number of instructions, interactions among tasks and amount of memory). Furthermore, the migration of tasks is performed only to a neighbor node (direct connection in the processors graph) if compared with the node that them reside currently. Du, Sun and Wu (DU; SUN; WU, 2007) model the processes migration on distributed, shared and heterogeneous environments. This model considers the states of the memory, of the process itself, of the I/O operations as well as network communication with other processes. Considering these parameters, the authors decide a new destination to a migratable process. Nevertheless, the authors do not specify either when the migration takes place or how the processes are elected for migration. Finally, considering the spectrum of related work presented in this chapter, we may conclude that there are not inittiatives that work in the communication and computation parts of an application and consider the migration costs when transferring processes.

After observing the related work described in this chapter, we can indicate some points that may represent limitations and future research opportunities. Formerly, some work are based on the assumption that the total application workload is know in advance (in compilation time) (GODFREY; KARP, 2006; ELSASSER; MONIEN; PREIS, 2000; SPOONER et al., 2003; HEISS; SCHMITZ, 1995). However, sometimes it is not possible to obtain the application composition previously (mainly at middleware level). Secondly, the use of a minimum percentage of gain to effectivate the migration is fixed in the following works (VADHIYAR; DONGARRA, 2005a; MORENO-VOZMEDIANO; ALONSO-CONDE, 2005). The former use 30% while the other use 20% for the percentage value. In this same context, Vadhiyar and Dongarra set the value for the migration costs in 900 seconds. This approach is not the better one in dynamic and non-dedicated environments, where fluctuations on network bandwidth and on machines utilization are common. Furthermore, a fixed period for rescheduling launching is demonstrated in the following approaches (HERNANDEZ; COLE, 2007; MONTERO; HUEDO; LLORENTE, 2003; UTRERA; CORBALAN; LABARTA, 2005). Lastly, some work require explicit calls for rescheduling in the application code directly. This is observed in (BHANDARKAR; BRUNNER; KALE, 2000) through calls to MPI_Migrate() directive. Thus, this mechanism implies in modifications in application code and in the fact that the programmer/user must know the behavior of her/his application deeply.

The application model chosen to construct our rescheduling model is BSP. This model is attractive for dynamic and heterogeneous environments. In this way, there are two approach aspects presented in the literature (SONG; TONG; ZHI, 2006): (i) develop a platform that supports BSP applications execution on dynamic and heterogeneous environments and; (ii) apply modifications on BSP model in order to fit it to a target environment. BSP-G is an example of the first approach. This BSP implementation does not change the concept of the model. It uses the Globus toolkit to execute the grid services. On the other hand, models like HBSP and DynamicBSP are examples of BSP extensions that present adaptations for heterogeneous computing, case of the first model, and for fault tolerance, case of DynamicBSP.

Considering the BSP applications context, we can present two initiatives for processes migration. The first one describes the PUBWCL (*Paderborn University BSP-Based Web Computing Library*) library (BONORDEN; GEHWEILER; HEIDE, 2005). PUBWCL takes profit from idle computing power of computers around the world using Internet links. The load balancing algorithms can perform processes migration during the superstep execution as well as at its end. They use information like the mean conclusion times of each superstep. Other work includes an extension of PUB library to support processes migration (BONORDEN, 2007). As the previous work, this also considers processors and computing times features. Furthermore, this Bonorden's work just include migration at the end of superstep, where the impact of the modification will be observed in the next one. The strategies implemented in (BONORDEN, 2007) take into consideration neither the processes communication nor the migration costs.

# 4 MIGBSP: PROCESSES RESCHEDULING MODEL

Considering the related work mentioned on last chapter, we developed a model for processes rescheduling called MigBSP. MigBSP acts over BSP (Bulk Synchronous Parallel) applications, controlling the migration of processes to new resources during their executions. Its final objective is to decrease the load unbalancing among processes, leveling and decreasing the time of their executions at each superstep. Aiming to reach this objective, we analyze data from both communication and computation parts of each BSP superstep as well as information related to migration costs.

This chapter presents MigBSP and its new ideas for processes rescheduling on BSP applications. We organized it as follows. Section 4.1 shows in details the motivation to develop MigBSP. This section also presents some issues related to processes migration on BSP applications. Section 4.2 presents our proposal of rescheduling model. Section 4.3 describes both the model of communication and parallel machine that will be used in the model. The definition of MigBSP is detailed in Section 4.4. Section 4.5 treats about the moment to trigger the processes rescheduling. Section 4.6 shows the algorithms used to choose the candidate processes for migration. Section 4.7 describes how the destination processor for a candidate processes for migration is selected. In addition, this section describes the considered calculus to measure the transferring viability of candidate processes. Section 4.8 is responsible to show the parameters of the model. Section 4.9 performs a theoretical analysis of the load balancing employed by MigBSP. Finally, Section 4.10 summarizes the chapter and emphasizes the main contributions of MigBSP.

## 4.1 Motivation

As we introduced in Chapter 1 and based on the work opportunities described in Section 3.5, we summarize our motivation to develop the present thesis in the following way. The remaining of this section will explain in details the motivation and important decisions for developing our BSP processes rescheduling model.

- Launch processes rescheduling automatically, without changes in application code and without any previous knowledge about both application and execution infrastructure;
- Use of relevant data and profit from BSP structure to decide the candidate processes for migration;
- Observation of the variance on processes behavior and on infrastructure data during application runtime in order to choose the candidate processes for migration and to launch the rescheduling mechanism at sliding intervals of supersteps.

First of all, we intend to develop a model for processes rescheduling that works automatically and without previous knowledge of the application behavior. The term automatic means that the user/programmer will not put explicitly calls for processes rescheduling inside her/his application. While the application is running, the model collects data about the processes and the execution infrastructure. These information will indicate the next rescheduling invocation transparently. Figure 4.1 illustrates our idea regarding the use of the model. The user/programmer does not need to change any line of code in her/his application. He/she compiles the application with an adapted library that uses both the implementation of the rescheduling model and a specific library used for BSP programming. The result is a binary that can be executed in a distributed system together with the implementation of the model. Lastly, our motivation consists in executing BSP applications faster without preliminary executions. Our mechanism represents an effortless manner to try performance improvements in BSP applications. Thus, the user/programmer may concentrate herself/himself especially on application development.

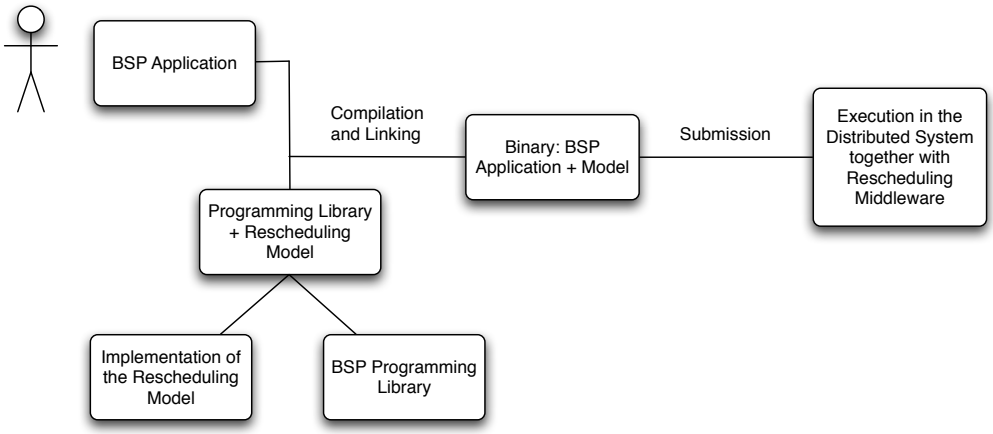


Figure 4.1: Model idea: from the application compilation up to its execution in the distributed system

Other relevant idea of the rescheduling model refers to the treatment of pertinent information for processes migration on dynamic and heterogeneous environments. Firstly, the main metric to perform load balancing in distributed systems is the computational load of the nodes or the computational time spent by each process/task to execute a set of instructions. Aiming to demonstrate a possible problem in this approach, we created an hypothetical infrastructure with two clusters and a migration situation between them. We modeled two clusters as Cluster1 and Cluster2. Both have 10 nodes, each one with one processor. In addition, each cluster has Gigabit Ethernet connection for communication intra cluster. The connection between them comprises links through the Internet with mean capacity of 10 Mbits/s. Cluster1 has nodes with 500 MHz while the second one presents nodes which have capacity of 1 GHz.

Suppose that the initial processes-resources assignment maps all 6 application processes on Cluster1. Each process is mapped to a different node. Concerning this, we can design a possible scenario of a computation-based processes rescheduling where process $p1$ is chosen for migration from cluster Cluster1 to Cluster2. This decision takes into consideration that this process is the slowest one and can run two times faster on Cluster2. Figure 4.2 depicts both situation: (i) before rescheduling of $p1$ and; (ii) after the migration. We can observe that scenario 4.2(b) resulted in a slower application,

since all communications to process $p1$ must go through the low bandwidth link that exists between the clusters. Therefore, it is important that load balancing and rescheduling strategies for grid and multi-cluster environments consider communication speed between the sites. This can lead in better decision when considering this kind of environment.

Besides computation, other possibility is apply load balancing through processes relocation taking into account only the communication metric. Now, suppose that $p1$ is running on Cluster2 while the other processes remain on cluster1. Moreover, we have Fast Ethernet link between the clusters in this context. This situation is revealed in Figure 4.3 (a). If a migration of $p1$ from Cluster2 to Cluster1 takes place, we will verify that its time to finish the computation phase will double if it is compared with the initial situation. Although the communications become faster, the performance of $p1$ will compromise the application time as a whole. It is simple to observe that communication based load balancing in heterogeneous systems must be accompanied with other techniques to fully exploit the characteristics of this environment.

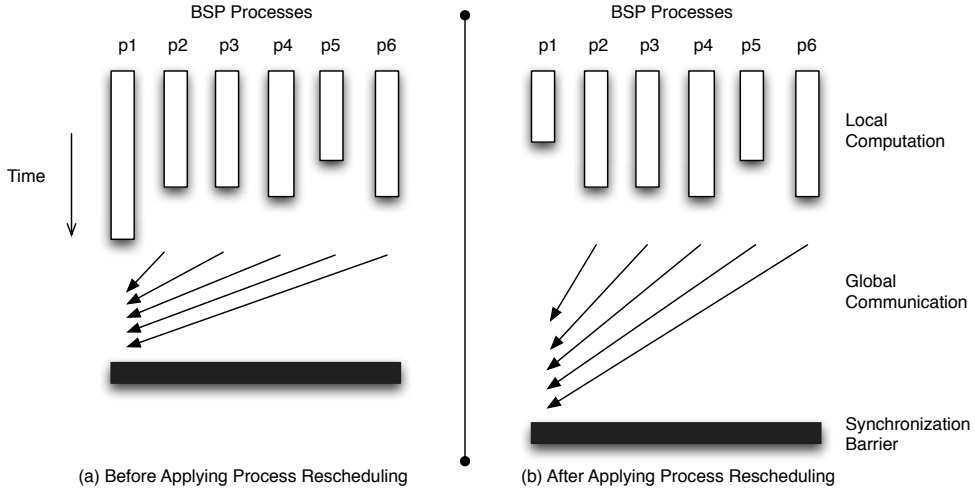


Figure 4.2: Migration of process $p1$ from cluster Cluster1 (slower) to Cluster2 (faster) considering a network with low bandwidth between them

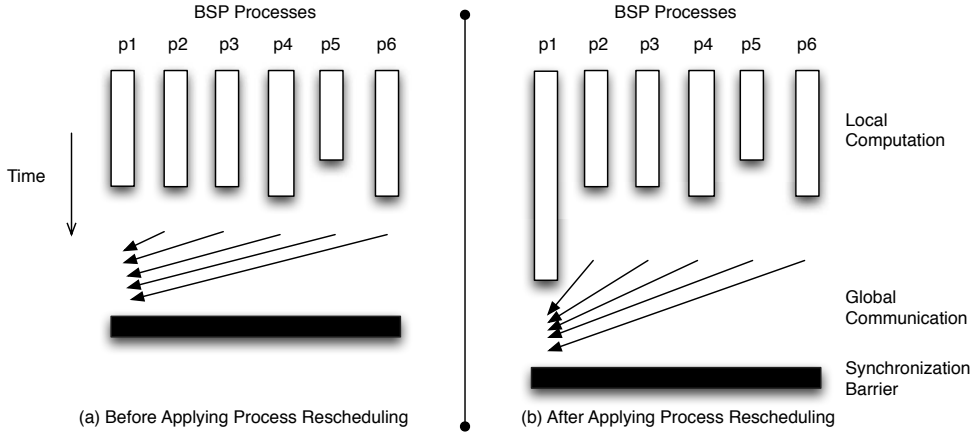


Figure 4.3: Migration of process $p1$ from Cluster2 (faster) to Cluster1 (slower) considering a network with high bandwidth between the clusters

A more refined load balancing approach can employ both computation and communication strategies on its decision of which processes should migrate. For example, a

process that is running in an overload processor can migrate to lightly loaded one that belongs to a cluster in which the considered process establishes high communication patterns. Concerning this, it is possible to reduce the superstep time on heterogeneous systems through the adjustment of both computation and communication phases. Despite this is true, processes migration can become sometimes the bottleneck for executing applications as depicts Figure 4.4. This situation occurs if the gain with processes rescheduling is lower than the migrations costs. The migration costs are related to the amount of virtual memory of the process, the migration of any specific data needed by the process, operating system overhead, the latency and the bandwidth between two endpoints and the costs related to the migration tool. This last item can cover checkpoint treatment, data serialization, reconstruction of the communication channels and so on.

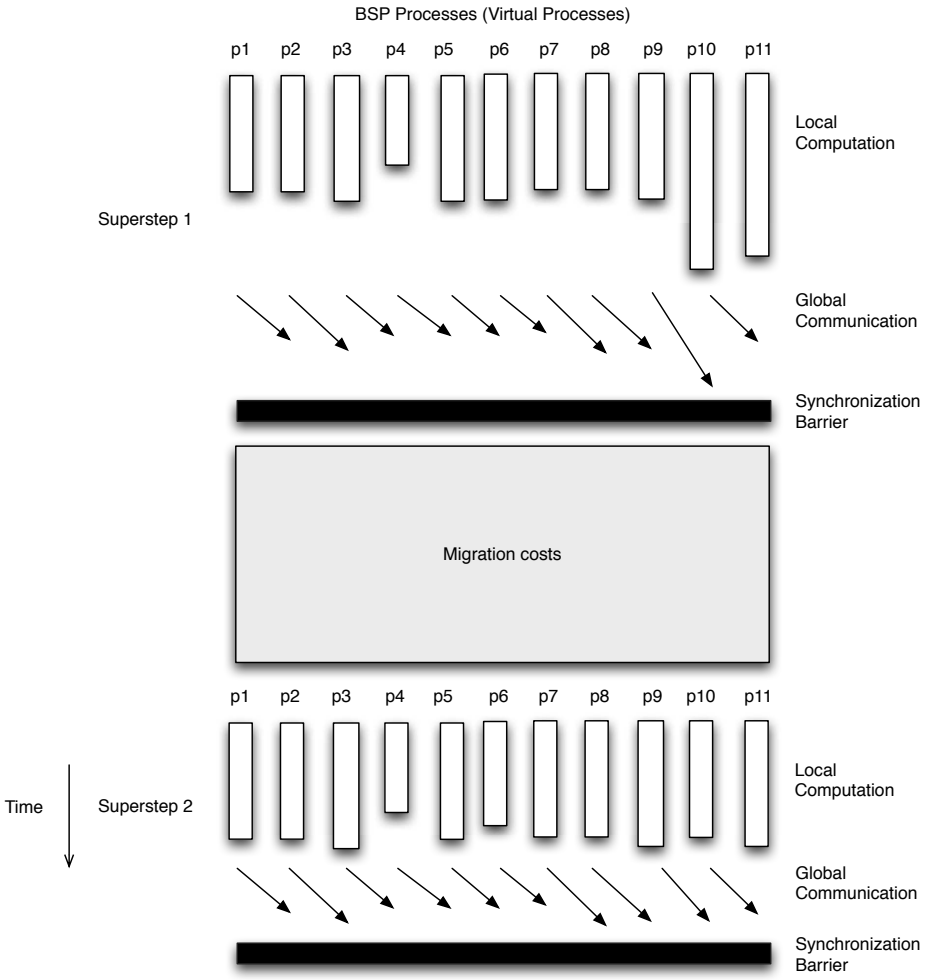


Figure 4.4: Communication and computation-based processes rescheduling with high migration costs

Figure 4.4 illustrates a situation where $p10$ and $p11$ are previously executing in a slower cluster, while the other processes are running in a faster cluster. The migration model decides to migrate $p10$ and $p11$ to the cluster which the remaining processes belong to. This target cluster presents more computing power. Analyzing the situation of Figure 4.4(b) where processes relocation reduces superstep time but causes large overhead, it is possible yet to achieve final improvements in application time. In this case, we will need a more elevated number of supersteps in order to amortize the impact of the migration

costs. Thus, the number of supersteps to be executed after performing migrations is a relevant factor when considering BSP processes rescheduling.

Finally, other important issue on BSP processes rescheduling is the model adaptation to changes on the processes' behavior and on the execution environment. Considering that a migration procedure may present high costs, it must be triggered according to the past behavior of the processes efficiently. Thus, we can predict their behavior on destination resources as a function of their past execution. For example, a process that presents a large variation in the number of instructions that executes at each superstep should be elected as a bad candidate for migration, because it should not recover the costs employed on its migration in performance improvements. Figure 4.5 shows a simple execution with three supersteps of a hypothetical BSP application. A rescheduling strategy can decide that process $p6$ is not a good candidate, because it presents large disparities in the amount of executed instructions. Here we suppose that the variation on $p6$ did not occur by any changes in resource's load.

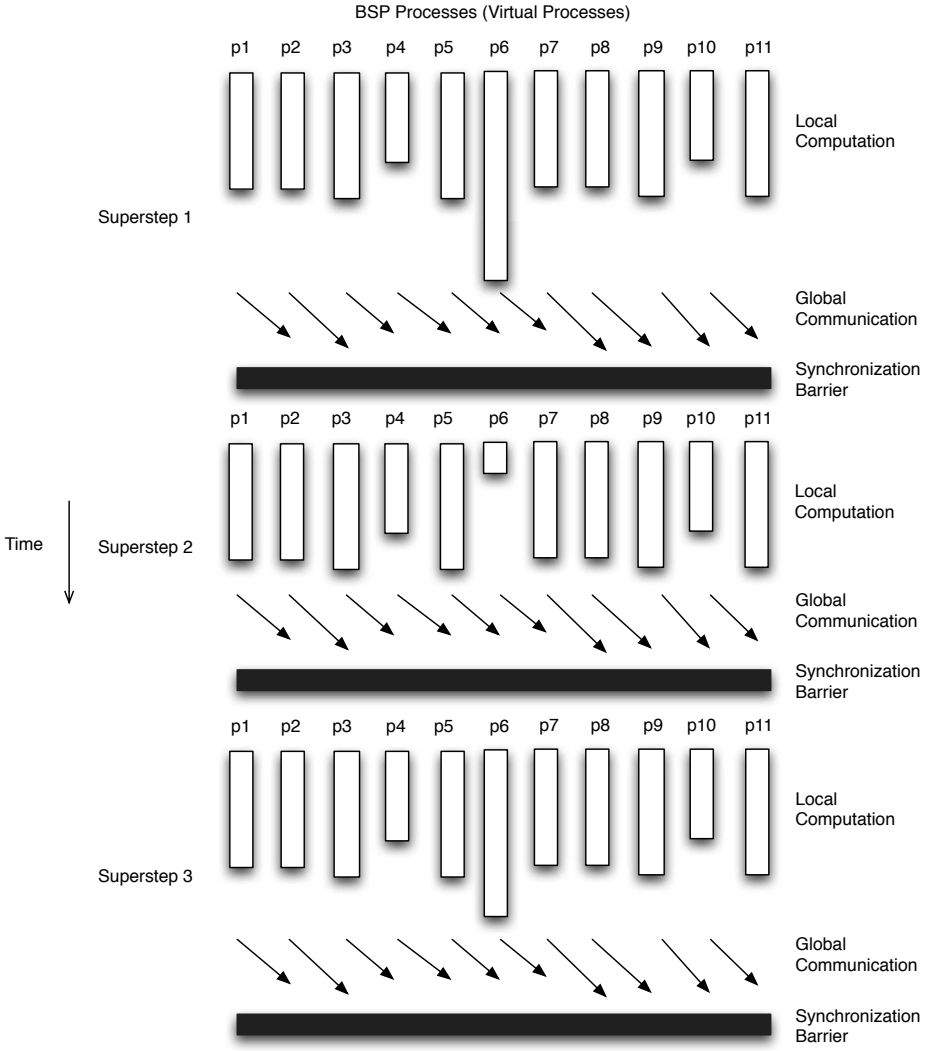


Figure 4.5: Observation of the processes' behavior in order to choose the candidates for migration

The organization in phases allows the detection of patterns of computation and communication easily. Moreover, it is possible to take profit from the BSP structure to launch

the procedure of processes rescheduling automatically. The mechanism of rescheduling at each superstep can be onerous and can become the bottleneck of the application execution. On the other hand, a strategy that uses thresholds for both computation and communication phases in order to control the gap between the fastest and the slowest times can be utilized on processes rescheduling at variable interval. Furthermore, a migration of only one process at each rescheduling launching cannot solve the performance problem, since the bottleneck can just pass from one process to another. Considering that a BSP application shows a communication behavior like demonstrated in Figure 4.5. In addition, the processes are mapped to 2 identical clusters and a low bandwidth is verified between them. Thus, the migration of only one process from Cluster2 to Cluster1 will not contribute for decreasing the application time. In this case, the bottleneck remains on inter-cluster communication. Depending on the resources availability on Cluster1, a better load balancing approach may occur by migrating all processes from Cluster2 to Cluster1.

## 4.2 Proposal

Concerning the motivations expressed above and the spectrum of work opportunities described in Section 3.5, we developed a model for BSP processes rescheduling called MigBSP (ROSA RIGHI et al., 2008; PILLA et al., 2009). The main objective of MigBSP is to reduce the application execution time through the control of processes migration between resources. The listing below presents the design decisions that guide MigBSP's development.

- MigBSP's work grain is a process of the operating system. A process is viewed as a region of code as well as allocated data up to the current execution point. MigBSP controls the processes relocation to different processors available in the considered distributed system. Furthermore, MigBSP does not deal with the initial processes-resources scheduling. This first mapping may be done arbitrarily and it is not addressed in the scope of this thesis;

- Processes rescheduling is performed in a dynamic manner, where necessary data for its functioning are captured during application execution. Moreover, MigBSP does not know either the number of instruction or communicated bytes at each superstep previously. Then, our migration model does not consider the total application workload, since it does not know this information;

- Both local computation and global communication phases of a BSP superstep are affected by MigBSP execution. The main idea of MigBSP is to reduce the length of the supersteps modifying the processes initial location and, thus, their respective computation and communication times;

- MigBSP controls the processes migration automatically without programmer and system administrator interventions. We are not using explicit function calls in application code. MigBSP acts at middleware level, not imposing modification on application level.

- The model for processes rescheduling was developed specifically for programs organized in synchronous phases, as suggest the execution rules of the BSP model.

We made our load balancing decision taking in mind the BSP organization and its synchronous character.

- The adaptation issue is pertinent for MigBSP model. In this context, we created two adaptations that act over the frequency of the rescheduling calls (see Subsections 4.5.1 and 4.5.2 for details). Both take profit from data collect during application runtime, aiming to reduce or extend the interval for rescheduling tests based on the system state: (i) processes are balanced or not and; (ii) if a pattern without migrations is described in the last attempts for processes rescheduling.

- Combination of multiple metrics to select the candidate processes for migration. The key idea behind this item is to consider relevant metrics for processes migration in distributed memory systems. We will analyze the metrics that act in favor of processes transferring as well as those that represent forces against migration. According to Du et al. (DU; SUN; WU, 2007), an appropriate rescheduling should consider the migration costs to avoid unproductive migrations.

- We prioritized the use of heuristics to choose the candidate processes for migration as well as the target processors for them. This decision was took in order to turn MigBSP competitive. MigBSP must handle fast scheduling algorithms in order to reduce its overhead on application execution. Our idea is to reduce the model's penalties when migrations are inviable (when migration costs are high, for instance).

- MigBSP verifies the regularity of the BSP processes when selecting them as good migration candidates or not.

Figure 4.6 (a) shows a superstep $k$ of an application in which the processes (load) are not balanced among the resources. Figure 4.6 (b) depicts the expected result with processes redistribution at the end of superstep $k$, which will influence the execution of the following supersteps ($k+1$ and so on). MigBSP provides a mathematical formalism that answers the following issues regarding processes migration and load balancing: (i) "When" to launch the processes migration; (ii) "Which" processes are candidates for migration and; (iii) "Where" to put an elected process from the candidates ones. MigBSP does not address the keyword "How", that treats the mechanism used to perform migrations.

MigBSP can be seen as a processes rescheduling model. In this way, according to distributed systems scheduling taxonomy of Casavant and Kuhl (1988), the present model can be enclosed on dynamic and global items. The dynamic item considers that information for processes scheduling are collected at application runtime. The role of scheduling is spread among the several processes that cooperate among themselves in order to improve resource utilization (processors and communication network). Thus, the model performs a physically distributed and cooperative scheduling. The achieved scheduling is sub-optimal and employs heuristics. Finally, following the horizontal classification of Casavant and Kuhl, the idea is to present an adaptive scheduling that can change its execution depending on the environment feedback.

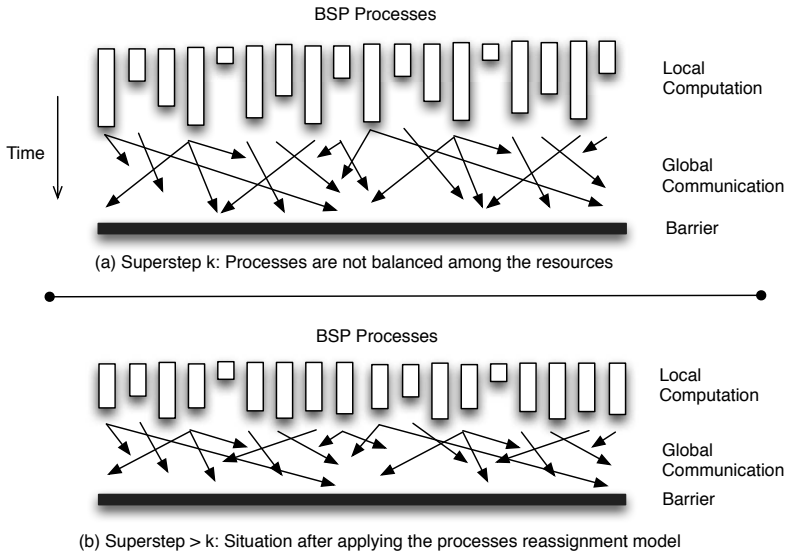


Figure 4.6: Observation of supersteps in different situations

## 4.3 Models of Parallel Machine and Communication

The BSP processes rescheduling model works over an heterogeneous and dynamic distributed environment. The heterogeneous issue considers the processors' capacities (all processors have the same machine architecture), as well as network bandwidth and level (Fast and Gigabit Ethernet and multi-clusters environments, for instance). The dynamic behavior deals with environment changes occurred at application runtime (such as network congestion and fluctuations on processors' load). Moreover, the dynamic behavior can also occur at process level, owing to some BSP processes may need more computing power and/or increase their network interaction with other processes during application execution.

Considering the paragraph above, we might say that MigBSP works over grids. Thus, the model of machine can include single-processor and multiprocessors machines, local networks as well as clusters. It is assumed that the parallel machine system consists of a collection of Sets $S$ connected by a communication network, as shown in Figure 4.7. A Set might be a LAN network or a cluster. Each Set contains one or more nodes. Lastly, each node offers one or more processors for BSP processes execution. Each processor may have different processing power. Moreover, we do not assume variations in the number of processes and processors. Besides this, our model of grid does not work with faults in both processes execution and distributed infrastructure (network and processors) perspectives. We do not apply restrictions in the number of administrative domains. The model may join wide area networks where the communication latency is known costly. The main idea of our machine model is to aggregate the processing power of machines already present in one institution or in institutions that describe cooperative work.

The use of Sets, nodes and processors classify MigBSP scheduling as hierarchical. We are basing our concepts on a notion of hierarchy (local and global) that is presented in the InteGrade scheduler (GOLDCHLEGER et al., 2004). The vision of hierarchy is important to optimize the passage of scheduling information. Besides nodes, a Set encompasses a Set Manager. The scheduling mechanism is located in every BSP process (additional code in barrier function) and in each Set Manager. This last entity captures data of a specific Set and exchanges them among other managers. We may observe the used hierarchical

notion in the example illustrated in Figure 4.8.

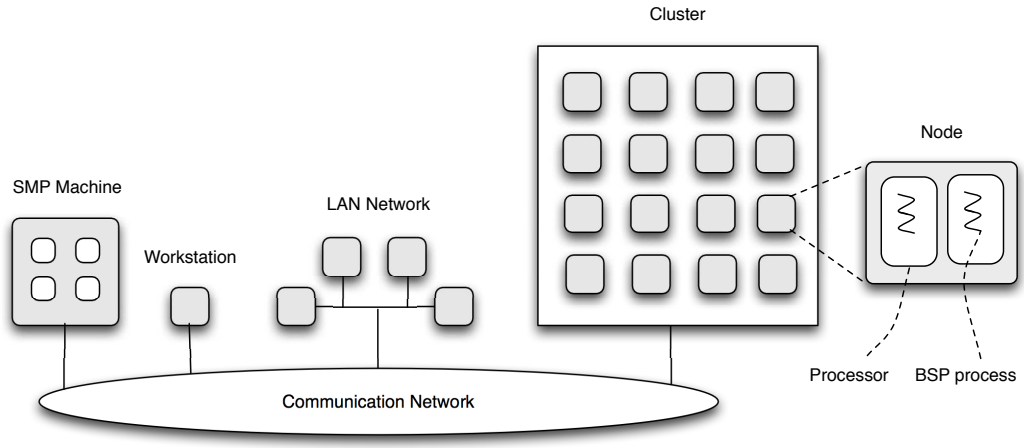


Figure 4.7: Model of Parallel machine with all-to-all asynchronous communication

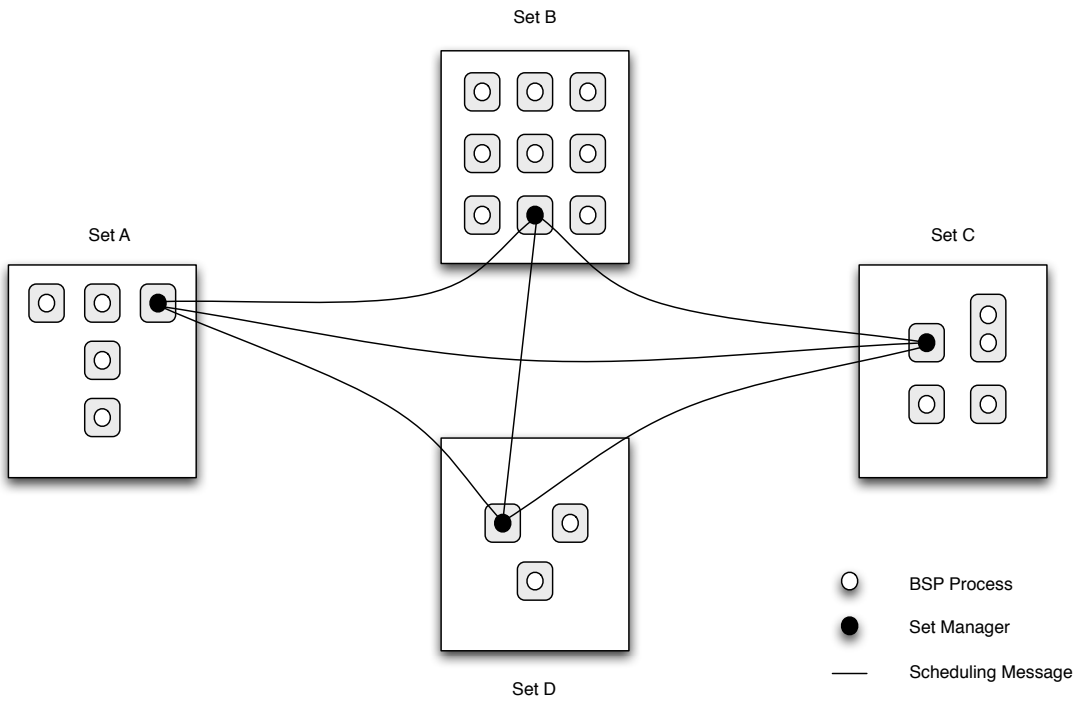


Figure 4.8: Instance of the hierarchical notion with Sets and Set Managers abstractions

Our communication model affirms that the Sets are fully interconnected, meaning that there exists at least one communication path between any two nodes. The only way for inter-node communication is through message passing. This communication is asynchronous, where the sending is non blocking while the receiving is blocking. Here, it is important to emphasize another restriction over the meaning of grid in the context of MigBSP. The underlying network protocol always guarantee that messages sent across the network are received in the order sent. In addition, the machine model guarantees that a message sent by one endpoint is received by another one since a link exists between them. Moreover, there is no efficient broadcasting service available and any link connection is represented by a bandwidth.

BSP model obligates asynchronous communication among the processes (SKILLICORN; HILL; MCCOLL, 1997). Considering this, we have the possibility to offer this facility through either asynchronism on both sending and receiving directives or only over the sending one. We opted for the last approach, since we can capture the communication time on the blocking receiving directive. Its implementation may take the initial and the final time for transferring a message to a destination process and may capture the interaction pattern to the target Set that this process belongs to. On the other hand, other possibility could be offer both communication directives in an asynchronous fashion. In this context, the first non-blocking communication call could trigger the initial communication time. In addition, we can wait for completion of all pending communications and take the final time in the beginning of the barrier function. The problem of this strategy is the fact that it considers the whole communication time. Using it, we cannot measure the interaction for different Sets when considering more than 2 processes in the system.

## 4.4 MigBSP Definition

Firstly, each BSP process is mapped to a physical processor that belongs to a node in the distributed system. The treatment of this first mapping is not addressed in this thesis. The produced scheduling at the beginning of superstep $k$ (identified as $S_k$) may be densely represented in a two dimension codification scheme (see Figure 4.9). The rescheduling problem is defined by the reallocation tests of a collection of $a$ processes ($p = \{p0, p1, p2, \ldots, p(a-1)\}$) to a set of $b$ processors ($P = \{P0, P1, P2, \ldots, P(b-1)\}$). In the codification of Figure 4.9, $S_k$ means the distribution of the BSP processes to processors during the $k^{th}$ superstep. In this representation, $M_{i,j}$ explains the mapping of process $j$ to processor $i$. If $M_{i,j}$ is equal to 1, we can affirm that $pj \in Pi$ and if it is equal to 0, $pj \notin Pi$ . A process belongs only to one processor during a specific superstep. In other words, each column of the two-dimension representation presents an unique value 1. On the other hand, a processor can execute more than one process. Therefore, each line may show more than once the value 1.

$$S_k = \begin{array}{c|ccccc} & p0 & p1 & \cdots & p(a-2) & p(a-1) \\ \hline P0 & M_{0,0} & M_{0,1} & \cdots & M_{0,a-2} & M_{0,a-1} \\ P1 & M_{1,0} & M_{1,1} & \cdots & M_{1,a-2} & M_{1,a-1} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ P(b-2) & M_{b-2,0} & M_{b-2,1} & \cdots & M_{b-2,a-2} & M_{b-2,a-1} \\ P(b-1) & M_{b-1,0} & M_{b-1,1} & \cdots & M_{b-1,a-2} & M_{b-1,a-1} \end{array}$$

Figure 4.9: Representation of the scheduling $s_k$ during the execution of superstep $k$

$$MigBSP = \{S, N, P, BSP, \lambda, \sigma, S_1, NS, CN, I\}.$$

MigBSP model is defined by the *MigBSP* notation expressed above. $S$, $N$, $P$ and *BSP* represent the Sets, the nodes, the processors and the BSP processes. $\lambda$ means the mapping of nodes to Sets. $\sigma$ is the mapping of processors to nodes. $S_1$ indicates the first processes-processors scheduling, i.e., the scheduling that will be used to execute superstep 1. *NS* defines the set of network segments (one hop link) that will be used to compose the topology. *CN* is a set which each element is composed by one or more network segment and two nodes. Thus, *CN* shows all connections between any two nodes

in our architecture. Consequently, we can have the connections among the processors. Lastly, $I$ represents the collection of initial parameters of our rescheduling model. This collection will be explained during the current chapter and summarized in Section 4.8.

## 4.5 Decision About Rescheduling Launching: "When"

The decision for processes remapping is taken at the end of a superstep (after barrier synchronization and before the next superstep). We are employing the reactive migration approach (MILANéS; RODRIGUEZ; SCHULZE, 2008), where the processes relocation is launched from outside the application transparently (in this case, at middleware level by MigBSP). The adopted migration point was chosen because in this moment it is possible to analyze data from all BSP processes at their computation and communication phases. In other words, at this point we have information about the slowest process, the amount of instructions performed by each one and the communication scheme among the processes. Aiming to generate as less intrusion in the application as possible, we applied two adaptations. They provide an adaptable interval between two calls for processes rescheduling. Both adaptations are discussed in the following subsections.

Each end of barrier synchronization represents a global consistent state. Besides a good point for migration launching, the end of barrier represents a pertinent time to set checkpoints for fault-tolerance (MIAO; TONG, 2007). Because an application is a sequence of supersteps, if any fault occurs in a certain superstep, the application needs to rollback for only one superstep. Therefore, this migration point was also chosen owing to it facilitates the processes transferring implementation. A local checkpoint can indicate the state of a candidate process for migration and may be sent from the source to the target processor in migration moment.

### 4.5.1 First Adaptation: Controlling the Rescheduling Interval based on the Processes Balance

To turn viable the adaptivity on processes rescheduling calling, it was used an index $\alpha$ ($\alpha \in N^*$) which informs the used interval of supersteps to apply the processes migration. This index increases if the system tends to the stability in conclusion time of each superstep and decreases on the contrary. The last case means that the frequency of calls increases to turn the system more stable quickly. In order to allow a sliding $\alpha$, it is necessary to verify if the distributed system is balanced or not. To treat this issue, the time of each BSP process is collected at the end of every superstep. Thus, the times of the slowest and the fastest processes are captured, and an arithmetic average of times is computed. Using these three values, it is possible to measure the balance among the BSP processes.

The distributed system is considered stable if both Inequalities 4.1 and 4.2 are true. In both inequalities, $D$ informs the percentage of how far the time of the slowest and the fastest processes can be from the average. $D$ value is passed in model initialization. Figure 4.10 reveals our idea to define balancing (stable) and unbalancing situations graphically. Algorithm 3 shows how the $\alpha$ value is computed during application runtime. Another variable called $\alpha'$ was employed to save the temporary value of $\alpha$. Thus, $\alpha'$ will indicate the next superstep interval to activate the load balancing. $\alpha'$ suffers a variation of one unity at each superstep depending on the state of the system.

$$time\ of\ the\ slowest\ process < average\ time\ .\ (1+D) \tag{4.1}$$

$$time\ of\ the\ fastest\ process > average\ time\ .\ (1-D) \tag{4.2}$$

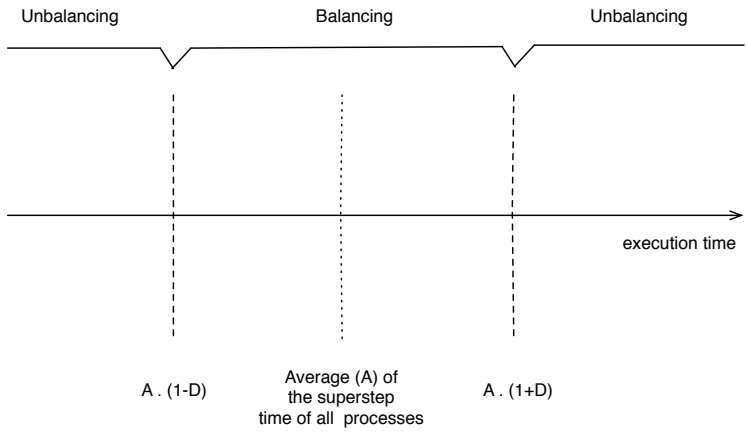


Figure 4.10: Analysis of both balancing and unbalancing situations which depend on the distance $D$ from the average time $A$

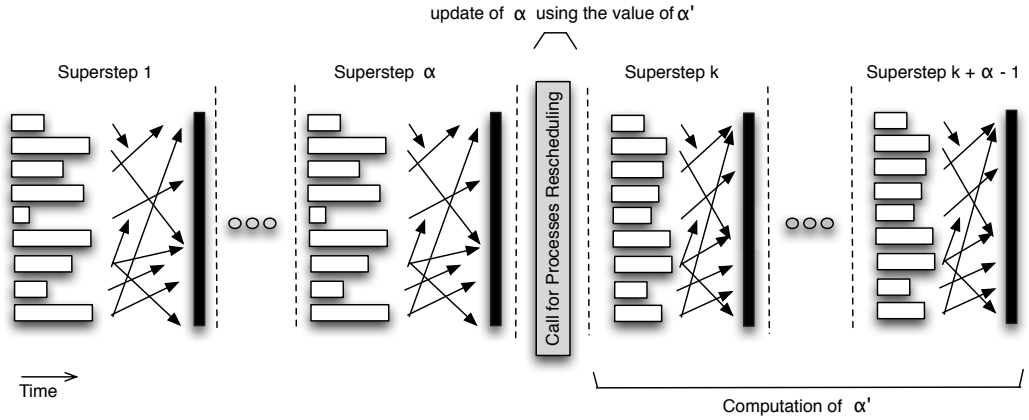


Figure 4.11: Overview of BSP application execution with MigBSP

In Algorithm 3, $t$ ($k \leq t \leq k+\alpha-1$) is the index of superstep and $k$ represents the superstep that comes after the last call for load rebalancing or it is 1 when the application is beginning ($k$ and $\alpha$ will have the same meaning in all algorithms). $\alpha'$ does not have an upper bound, but its lower value is the initial value of $\alpha$. The idea of the model is to minimize its intrusion in application execution while the system stays stable, postponing the processes rescheduling activation according to $\alpha$. Figure 4.11 illustrates the execution line of a BSP application. $\alpha'$ is computed according data collected by each process between two calls for BSP processes rescheduling.

In implementation view, BSP processes save their superstep time in a vector and pass it to their Set Managers when rescheduling is activated. Following this, all Set Managers exchange their informations. Set Managers have the times of each BSP process and compute both Inequalities 4.1 and 4.2. Therefore, each manager knows the $\alpha'$ variation locally.

Every load balancing strategy must answer the question about the correct moment to come into operation. We are using $\alpha$ index to control this moment, which could be computed internally or externally to MigBSP. We opted to compute $\alpha$ internally to MigBSP

**Algorithm 3** Computation of $\alpha$

```
1: for t from superstep k to superstep k + α − 1 do
2:    if Inequalities 4.2 and 4.1 are true then
3:       Increase α′ by 1
4:    else if α′ > initial α then
5:       Decrease α′ by 1
6:    end if
7: end for
8: Call for BSP processes reassignment
9: α = α′
```

for simplicity mainly. We took profit from BSP organization to save BSP processes information without extra costs. We know all data about both computation and communication phases of a BSP superstep at barrier moment. Thus, we analyze these data for scheduling calculus. On the other hand, monitoring tools like Ganglia and Monalisa could be used to compute $\alpha$ externally. However, we pretermited this option since these tools must be supplied with data that we already know at middleware level. Furthermore, additional communications and synchronizations between BSP processes and the monitoring tool lead to costs that we can avoid.

### 4.5.2 Second Adaptation: Controlling the Rescheduling Interval based on the Number of Calls without Migrations

Other adaptation considers the management of $D$ (see Inequalities 4.1 and 4.2) based on the frequency of migrations. The idea is to increase $D$ if processes rescheduling is activated for $\omega$ consecutive times and none migrations happen. The increase of $D$ enlarges the interval in which the system is considered stable as well, causing the increase of $\alpha'$ consequently. On the other hand, $D$ can decrease down to a limit if each call produces the migration of at least one process. Algorithm 4 presents how $D$ is controlled at each rescheduling call.

**Algorithm 4** Stability of the system according to $D$

```
1: γ ← Consecutive rescheduling calls without migrations
2: if γ ≥ ω then
3:    if D + D/2 < 1 then
4:       D ← D + D/2
5:    end if
6: else if D > initial D and γ = 0 then
7:    D ← D - D/2
8: end if
```

$D$ computation is done by each Set Manager, which knows if migrations occurred or not during the call for processes rescheduling. This adaptation is important when the migration costs are high. Thus, although a process is selected for migration, its transferring will not take place and the system will remain with the same processes-resources configuration. Therefore, it is pertinent to increase $D$ (and consequently, the value of $\alpha$) in order to minimize MigBSP impact on application execution in this situation.

### 4.5.3 Analyzing the Number of Rescheduling Calls

Firstly, we will demonstrate the number of calls when the system remains stable during application execution, which characterizes the best case. Migrations performed in this situation do not change the system state, *i.e.*, they do not provide unbalancing among the processes' times. $\alpha$ doubles after each call for process rescheduling, providing a geometric progression with ratio 2. Equation 4.3 shows the value of $\alpha$ ($a_c$) when $c^{th}$ call for processes rescheduling takes place. Initial $\alpha$ is the first element in the progression. Therefore, we can obtain the number of supersteps to reach the $c^{th}$ rescheduling call as follows: $S = \sum_{i=1}^{c} a_c$. Adopting initial $\alpha$ 4, $\alpha$ will be equal to 32 when the fourth call for processes rescheduling occurs. Thus, the sum of $4+8+16+32$ represents the amount of supersteps until this moment.

$$a_c \ = \ \alpha \ . \ 2^{c-1} \tag{4.3}$$

In the worst case, the system begins unstable and $\alpha$ remains the same at each call for processes rescheduling (we are not considering the dynamic behavior in this analysis). During instability phase, $\alpha$ is constant ($a_c = \alpha$) and the number of supersteps at rescheduling call with order $c$ is $S = \sum_{i=1}^{c} a_c$. The number of supersteps can be modeled as an arithmetic progression with ratio $\alpha$. During this phase, we have three possibilities of progress when a call is achieved: (i) migrations are done; (ii) $\omega$ attempts for processes migrations are crossed without processes transferring; (iii) migrations do not take place and the amount of calls without migrations is lower than $\omega$. Both cases i and ii can make the system stable. In the second case especially, $D$ increases if a pattern with $\omega$ calls without migration occurs, increasing the stability tendency of the system. The third case means that the system remains unstable and the next call for processes rescheduling will be done at $\alpha$ next supersteps.

Considering an application with $s$ supersteps, calls for processes rescheduling will take place at least $l$ ($Max(l) \mid s \geq \sum_{i=1}^{l} \alpha.2^{i-1}$) and at most $\frac{s}{\alpha}$ times. For example, MigBSP will launch processes rescheduling at most 25 times and at least 4 times over a BSP application with 100 supersteps and initial $\alpha$ equal to 4. The choice of both $D$ and $\alpha$ parameters influence the behavior of the rescheduling model. The greater the value of $D$, the greater the possibility to establish a stable system. The appropriate value of $\alpha$ will depend on the application's behavior.

## 4.6 Decision About the Candidates for Migration: "Which"

The answer for "Which" is solved through our decision function called Potential of Migration (PM). Each BSP process $i$ computes $n$ functions $PM(i, j)$, where $n$ is the number of Sets and $j$ means a specific Set. The key idea to use this approach (processes and Sets) consists in not performing all available processes-resources tests at the rescheduling moment. $PM(i, j)$ is found through the combination of Computation, Communication and Memory metrics. As the own names suggest, the first two work at the computation and communication phases of a superstep, respectively. Memory metric acts in MigBSP as an idea of migration costs. It will indicate the process transferring viability. The result of each metric presents the same unit, enabling us to combine them properly. We will explain in details the PM computation and the metrics whose compose it in the following subsections.

### 4.6.1 Computation Metric

Each process $i$ computes $Comp(i, j)$ functions. $Comp(i, j)$ informs the Computation metric for process $i$ and a specific Set $j$. The data used to calculate this metric start at superstep $k$ and finish at superstep $k+\alpha-1$. The relation of these data are organized in the following manner: (i) computation time prediction of process $i$; (ii) Computation Pattern of process $i$ in order to observe the stability, or regularity, in the number of instructions during the supersteps crossing and; (iii) performance degree of Set $j$.

For every superstep $t$ ($k \leq t \leq k+\alpha-1$), the number of processor's instructions ($I_t$) and the time of conclusion of the computation phase ($CT_t$) are stored. The value of $I_t$ is used to evaluate the process stability (regularity), that is represented by the Computation Pattern called $P_{comp}(i)$. This pattern is a real number enclosed in an [0,1] interval. A $P_{comp}(i)$ close to 1 means that the process $i$ is regular in the number of instructions that executes at each superstep. On the other side, this pattern will be close to 0 if the process suffers large variations in the amount of executed instructions. Its initial value is 1 for all processes because it is made an assumption that all processes are stable. Logically, this value goes down if this bet is not proven.

$P_{comp}(i)$ of process $i$ increases or decreases depending on the prediction of the amount of performed instructions at each superstep. $PI_t(i)$ represents this prediction for superstep $t$ and process $i$. It is based on the Aging concept (TANENBAUM, 2003). Following this scheme, the prediction of instructions of a specific superstep depends on the data regarding itself and all previous supersteps up to the first one after the reassignment call. For instance, $PI_t(i)$ at superstep $K+3$ needs data from supersteps $k+3$, $k+2$, $k+1$ and $k$. The Aging concept uses the idea that the prediction value is more strongly influenced by recent supersteps. The generic formula to compute the prediction $PI_t(i)$ for process $i$ and superstep $t$ is shown below.

$$PI_t(i) = \begin{cases} I_t(i) & if\ t = k \\ \frac{1}{2}PI_{t-1}(i) + \frac{1}{2}I_t(i) & if\ k < t \leq k+\alpha-1 \end{cases}$$

The advantage of this prediction scheme is that only data between two processes reassignment activations (among the supersteps $k$ and $k+\alpha-1$) is used. This scheme saves memory and contributes to decrease the prediction calculation time. On the other hand, the value of $P_{comp}(i)$ persists during the BSP application execution independently of the amount of calls for reassignment. $P_{comp}(i)$ is updated following the Algorithm 5. We consider the system stable if the forecast is within a $\delta$ margin of fluctuation from the amount of instructions performed. For instance, if $\delta$ is equal to 0.1 and the number of instructions is 50, the prediction must be between 45 and 55 to increase the $P_{comp}(i)$ value.

**Algorithm 5** Computation Pattern $P_{comp}(i)$ of the process $i$

```
1: for t from superstep k to superstep k+α−1 do
2:   if PI_t(i) ≥ I_t(i).(1−δ) and PI_t(i) ≤ I_t(i).(1+δ) then
3:     Increases P_comp(i) by 1/α up to 1
4:   else
5:     Decreases P_comp(i) by 1/α down to 0
6:   end if
7: end for
```

The computation pattern $P_{comp}(i)$ is an element in $Comp(i, j)$ function. Other one is a computation time prediction $CTP_{k+\alpha-1}(i)$ of the process $i$ at superstep $k+\alpha-1$ (last

superstep executed before processes reassignment). Analogous to *PI* prediction, *CTP* also works with Aging concept. Supposing that $CT_t(i)$ is the computation time of the process $i$ during superstep $t$, then the prediction $CTP_{k+\alpha-1}(i)$ is computed as follows.

$$CTP_t(i) = \begin{cases} CT_t(i) & if\ t = k \\ \frac{1}{2}CTP_{t-1}(i) + \frac{1}{2}CT_t(i) & if k < t \leq k+\alpha-1 \end{cases}$$

Finally, $Comp(i,j)$ presents an index $ISet_{k+\alpha-1}$(j). This index informs the average capacity of performance of the Set $j$ at $k+\alpha-1^{th}$ superstep. For each processor in a Set, its load is multiplied by its theoretical capacity. Concerning this, Set Managers compute a performance average of their Sets and exchange this value. Each manager calculates $ISet(j)$ for each Set normalizing the performance average of each Set by its own average. In the sequence, all Set Managers pass $ISet(j)$ index to the BSP processes under its jurisdiction.

$$Comp(i,j) = W_{comp}\ .\ P_{comp}(i)\ .\ CTP_{k+\alpha-1}(i)\ .\ ISet_{k+\alpha-1}(j) \tag{4.4}$$

Equation 4.4 shows the function to calculate the Computation metric for process $i$ to set $j$. Coefficient $W_{comp}$ is a constant used to represent the weight of this metric. The equation value is high if the BSP process presents stability on its executed instructions, has a large computation time and an efficient Set is involved. However, $Comp(i,j)$ is close to 0 if the process is unstable and/or it finishes its computation phase quickly. The model aims to migrate a delayed BSP process that presents a good behavior (amount of instructions that performs is regular) on the resource which belongs currently, because it can follow this actuation in another resource. In addition, we are considering the target Set in order to evaluate its capacity to receive a process.

### 4.6.2 Communication Metric

Communication metric is expressed through $Comm(i,j)$, where $i$ denotes a BSP process and $j$ the target Set. This metric treats the communication (just receiving actions) involving the process $i$ and all processes that belong to Set $j$. One process may migrate to another processor that belongs to its same Set. In order to compute $Comm(i,j)$, data collected at superstep $k$ up to $k+\alpha-1$ are used. We present the information needed to compute this metric in the following itemization.

- Regularity regarding the number of received bytes by one process;
- Prediction time of data receptions of a process BSP in relation to different Sets.

Each process maintains a communication time for a specified Set at each superstep and a pattern of communication called $P_{comm}(i,j)$. This pattern is a real number within the [0,1] interval. Its alteration depends on the prediction $PB_t(i,j)$, which deals with the amount of bytes involved during receptions performed by process $i$ from sendings executed by processes that belong to Set $j$ at superstep $t$. Analogous to Computation metric, $PB_t(i,j)$ is based on Aging concept and is organized as follows.

$$PB_t(i,j) = \begin{cases} B_t(i,j) & if\ t = k \\ \frac{1}{2}PB_{t-1}(i,j) + \frac{1}{2}B_t(i,j) & if k < t \leq k+\alpha-1 \end{cases}$$

In $PB(i,j)$ context, $B_t(i,j)$ is a notation used to assign the number of received bytes by process $i$ at superstep $t$ from sendings of processes that belong to Set $j$. Algorithm 6 uses

this prediction to compute $P_{comm}(i,j)$. This algorithm uses a variable $\beta$ which informs the acceptable variation in communication prediction. Similarly to $\delta$, if $\beta$ 0.1 and $B_t i,j$ is 100, we must have our prediction between 90 and 110 in order to configure superstep $t$ as regular.

**Algorithm 6** Communication Pattern $P_{comm}(i,j)$

```
1: for t from superstep k to superstep k+α−1 do
2:   if (1−β).B_k(i,j) ≤ PB_k(i,j) and (1+β).B_k(i,j) ≥ PB_k(i,j) then
3:     Increases P_comm(i,j) by 1/α up to 1
4:   else
5:     Decreases P_comm(i,j) by 1/α down to 0
6:   end if
7: end for
```

$P_{comm}(i,j)$ is the first element in $Comm(i,j)$. The second one is communication time prediction $BTP_{k+\alpha-1}(i,j)$ involving the process $i$ and Set $j$ at superstep $k+\alpha-1$. In order to compute this prediction, the communication time of receivings $BT_t(i,j)$ from process $i$ of sendings from processes that belong to Set $j$ at superstep $t$ is used. Concerning this, $BTP_{k+\alpha-1}(i,j)$ is achieved as follows.

$$BTP_t(i,j) = \begin{cases} BT_t(i,j) & if \;\; t = k \\ \frac{1}{2}BTP_{t-1}(i,j) + \frac{1}{2}BT_t(i,j) & if \; k < t \leq k+\alpha-1 \end{cases}$$

$$Comm(i,j) = W_{comm} \; . \; P_{comm}(i,j) \; . \; BTP_{k+\alpha-1}(i,j) \tag{4.5}$$

The function that computes Communication metric is presented in Equation 4.5. Coefficient $W_{comm}$ informs the weight of Communication metric. The result of Equation 4.5 increases if the process $i$ has a regularity considering the received bytes from processes of Set $j$ and performs slower communication actions to this Set. The value of $Comm(i,j)$ is close to 0 if process $i$ presents large variations in the amount of received data from Set $j$ and/or few (or none) communications are performed with this Set. MigBSP does not take into consideration the number of messages, but the amount of bytes involved in communication (data receiving). Thus, the situation where 50 bytes are passed in 50 messages of 1 byte is equal to other where 50 bytes divided in two messages. Nevertheless, we know that in each communication interaction a latency is present to open the communication channel. The latency is not used as isolated parameter in MigBSP, but together with the communication time.

### 4.6.3 Memory Metric

Function $Mem(i,j)$ represents the Memory metric and evaluates the migration costs to transfer the image of process $i$ to a resource in Set $j$. This metric just uses data collected at the superstep in which the load rebalancing will be activated (where $\alpha$ is achieved). Memory Metric works with the following factors related below.

- The quantity of bytes in memory of each BSP process;
- Data transferring time between the Set Manager of the BSP process and the manager of the target Set;
- Costs relates to the adopted migration mechanism.

Firstly, the memory space in bytes of considered process is captured through $M(i)$. After that, the transfer time of 1 byte to the destination Set is calculated through $T(i,j)$ function. The communication involving the Set Manager of the process $i$ is established with the Set Manager of each considered Set $j$. Finally, it is calculated the time spent on migration operations in $Mig(i,j)$ function. These operations are dependent of the operating system, as well as the tool used to provide processes migration. They can include, for example, connections reorganizations, memory serialization, checkpoint recovery, time spent to create another process in the target host, and so on.

$$Mem(i,j) = W_{mem} \,.\, (\, M(i) \,.\, T(i,j) \,+\, Mig(i,j) \,) \tag{4.6}$$

Equation 4.6 shows the elements of $Mem(i,j)$. In this equation, $W_{mem}$ is a coefficient that refers to the weight of this metric. Analyzing Memory metric, each BSP process will compute $n$ times $Mem(i,j)$, where $n$ is the number of Sets in the environment. The lower the value of $Mem(i,j)$ the easier the transferring of process $i$ to Set $j$. On the other hand, as $Mem(i,j)$ increases, the migration costs of the process $i$ to Set $j$ grows as well.

### 4.6.4 Potential of Migration

Aiming to generate the Potential of Migration ($PM$) of each process, we use the notion of force from Physics. In Physics, force is an influence that can make an object accelerate and is represented by a vector. A vector has a size (magnitude) and a direction. Analyzing the force idea, each studied metric can be seen as a vector that acts over an object. In our case, this object is the migration of a BSP process. Vectors $\vec{Comp}$ and $\vec{Comm}$ represent the Computation and Communication metrics, respectively. Both vectors have the same direction and stimulate the process migration. On the other hand, the Memory metric represents the migration costs and is symbolized by vector $\vec{Mem}$. This vector works against the migration, since its direction is opposite to $\vec{Comp}$ and $\vec{Comm}$. Each vector is used to compute the resultant force. If two forces present the same direction, the length of the resultant force is the sum of the magnitude of the forces that act over the object. On the other hand, if the forces presents opposite direction, we subtract the smallest by the largest to measure the resultant force.

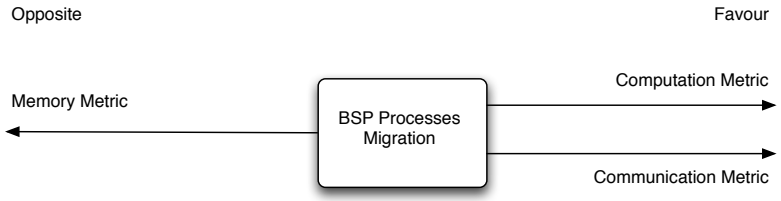


Figure 4.12: Computing the resultant force (Potential of Migration) considering three metrics: (i) Computation and Communication metrics act in favor of migration; (ii) Memory metric works in the opposite direction as migration costs

Figure 4.12 shows the actuation of Computation, Communication and Memory metrics to compute the Potential of Migration. In processes reassignment model context, the Potential of Migration of a process $i$ to Set $j$ is denoted by $PM(i,j)$ and is computed by Equation 4.7. $Comp(i,j)$, $Comm(i,j)$ and $Mem(i,j)$ represent the Computation, Communication and Memory metrics, respectively. Considering that $Comp(i,j)$ and $Comm(i,j)$ can have their values close to 0, then $PM(i,j)$ can receive negative values. The greater the value of $PM(i,j)$, the more prone the BSP process will be to migrate. A high $PM(i,j)$

means that process $i$ has high computation time, high communication with processes that belong to Set $j$ and presents low migration cost to this Set.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{4.7}$$

Each process $i$ will compute $n$ times Equation 4.7 locally, where $n$ is the amount of Sets in the environment. After that, process $i$ sends its highest Potential of Migration to its Set Manager. All Set Managers exchange their $PM$ values. Figure 4.13 presents an example with this message exchange among the Set Managers. Our strategy was based on the previous work of Shah et al (SHAH; VEERAVALLI; MISRA, 2007). However, they apply data exchange at each fixed period interval of time and this procedure involves just communication among all processes.

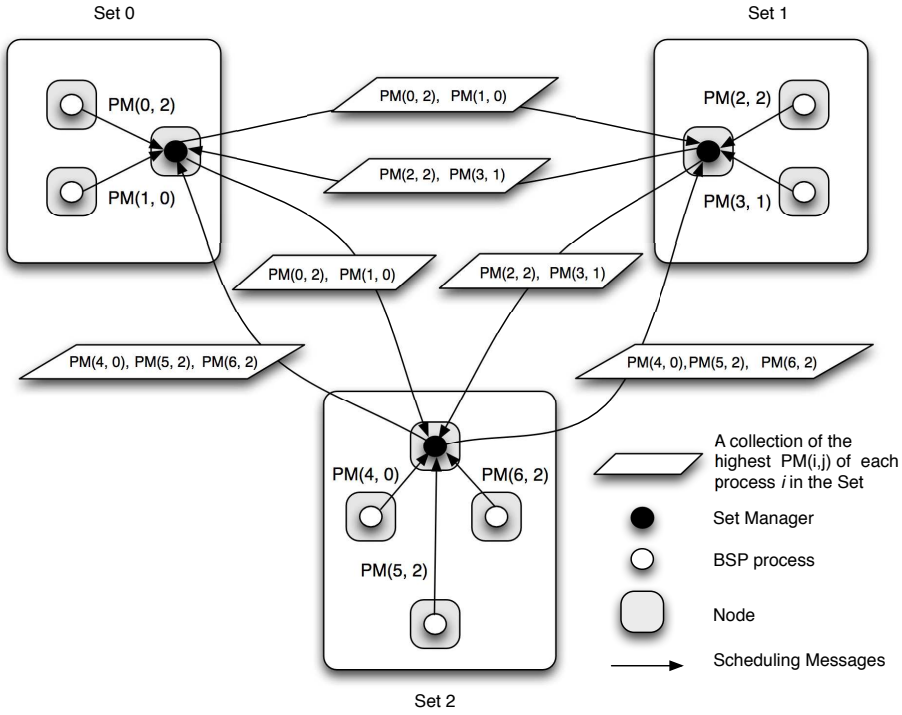


Figure 4.13: Message passing among the Set Managers with a collection of the highest $PM(i,j)$ of each BSP process

We applied list scheduling in order to select the candidates for migration. Each Set Manager creates a decreasing ordered list based on the highest $PM$ of each BSP process. Negatives values for $PM$ are discarded from the list. MigBSP uses this list to apply one of two possible heuristics in order to select the candidates processes for migration. The first heuristic chooses processes that have $PM$ higher than a $MAX(PM).x$, where $MAX(PM)$ is the highest $PM$ and $x$ a percentage. The second heuristic chooses one BSP process, the first of the list, whose has the highest $PM$ value. Each selected process is passed to the MigBSP's algorithm which treat the the election of a new target processor to execute it.

## 4.7 Decision About the Destination of Processes: "Where"

Process migration happens after the barrier synchronization of superstep which $\alpha$ is reached (see subsection 4.5). In this moment, all BSP processes are waiting for MigBSP migration actions before the beginning of the next superstep. An elected process $i$ has a

target Set $j$ informed on its Potential of Migration $PM(i, j)$. Thus, the pertinent question is to choose which the processor (and consequently the node) of this Set may be the destination of the process. Algorithm 7 lists the steps to answer the question "Where", which choose a target processor for each candidate process. Firstly, the Set Manager of process $i$ contacts the manager of the Set $j$ asking it for a processor to receive a process. This manager verifies the resources under its responsibility and elects the destination processor.

**Algorithm 7** Choosing a target processor to a candidate process $i$

1: Set Manager $SM_i$ signalizes process $i$ and informs it that it is a candidate for migration;
2: Set Manager $SM_i$ calls the destination Set Manager $SM_j$ and asks it for a processor;
3: Set Manager $SM_j$ selects a processor in its jurisdiction through an evaluation function;
4: Set Manager $SM_j$ returns information about the chosen processor to $SM_i$. The former also passes information regarding the execution time of process $i$ in this processor;
5: Set Manager $SM_i$ predicts the evaluation performance of process $i$, computes the migration gains and launches the process migration if it is viable.

The manager of the destination Set calculates the time which each processor takes to compute the work assigned to it. This is performed through Equation 4.8. $time(p)$ captures the computation power of processor $p$ taking into account the external load (processes that do not belong to BSP application). $load(p)$ represents the CPU load average on the last 15 minutes. This time interval was adopted based on work of Vozmediano and Conde(MORENO-VOZMEDIANO; ALONSO-CONDE, 2005). Equation 4.8 also works with instruction summing of each BSP process assigned to processor $p$ in the last executed superstep. In this context, $M(i, p)$ is equal to 1 if the process $i$ is executing on processor $p$. The processor $p$ with the shortest $time(p)$ is chosen to be tested to receive a BSP process. After that, this Set Manager computes Equation 4.9 based on data from process $i$, as well as from its own Set.

$$time(p) = \frac{\sum_{i,p:M(i,p)=1} I_{k+\alpha-1}(i)}{(1 - load(p)) \; . \; cpu(p)} \tag{4.8}$$

$$t1 = time(p) + B_{k+\alpha-1}(i, j) \; . \; T(i, j) \; + \; Mem(i, j) \tag{4.9}$$

The idea of Equation 4.9 is to simulate the execution of considered process in the destination Set taking into account the migration costs. In this situation, $time(p)$ is the simulation of the execution of process $i$ on target processor $p$. In the same way, $T(i, j)$ refers to the transferring rate of 1 byte of process $i$ inside the Set $j$ (communication established with the Set Manager). $Mem(i, j)$ is the Memory Metric and is associated with the migration costs ($W_{mem}$ equal to 1) . Contrary to $time(p)$ and $T(i, j)$, $Mem(i, j)$ involves the current location of process $i$ and target Set $j$. The manager of Set $j$ sends to the manager of process $i$ the destination processor $p$ and $t1$ value.

$$t2 = time(p') + B_{k+\alpha-1}(i, j) \; . \; T(i, j) \tag{4.10}$$

The actual Set Manager of process $i$ computes Equation 4.10 which aims to analyze the execution of the process considering its current location. In this situation, $p'$ is the current processor of process $i$ and $T(i, j)$ means the transfer rate between the current Set Manager of process $i$ and the target manager of Set $j$. In both Equations 4.9 and 4.10,

$B_{k+\alpha-1(i,j)}$ refers to the number of received bytes of process $i$ from sendings of processes that belong to Set $j$ at superstep $k+\alpha-1$. Process $i$ will migrate from $p'$ to $p$ only if $t2 > t1$. This migration approach was based on a previous work from Du, Sun and Wu (2007).

## 4.8 Model Parameters

Besides the number of processes, processors, Sets and the first processes-resources mapping, MigBSP needs a collection of parameters to initialize its functioning. Considering this, we present them using the following notation expressed below.

$$I = \{D, \alpha, \delta, \beta, W_{cmp}, W_{com}, W_{mem}\}.$$

For instance, the set $I = \{0.2, 10, 0.1, 0.2, 1, 1, 1, 0.3\}$ means that MigBSP works with $D$ equal to 0.2. Thus, the times of the slowest and the fastest processes must be at most 20% away from the average time to consider the processes as balanced. The value 10 is passed for the initial $\alpha$. This parameter is responsible to control the interval of supersteps for the next processes rescheduling call. $\alpha$ is updated at each rescheduling activation and will indicate if the processes are in time equilibrium or not.

The third and the fourth elements in $I$ receive the values 0.1 and 0.2. The former refers to $\delta$, which acts on Computation metric. $\delta$ is used to to verify if the prediction of instruction is close or not to the real number of executed instructions. Therefore, this parameter is used to increase or decrease the value of Computation Pattern of each BSP process. The prediction of instructions should be at most 10% above or below from the number of instruction to characterize a hit. The value 0.2 is passed to $\beta$ and its management is analogous to $\delta$ for Communication metric. If the amount of bytes received by process $i$ from sendings of processes that belong to Set $j$ is 200, then the communication prediction must be between 160 and 240 in order to increase the value of the Communication Pattern of process $i$ related to Set $j$. The following three parameters describe the weight of the studied metrics. Considering that each metric may be viewed as a force that acts over the process migration, each metric weight may represent the angle of force incidence. Thus, the value 1 could be the highest while 0 the lowest.

## 4.9 Load Balancing Decisions Analysis

Section 2.2 in Chapter 2 presented a study about load balancing. In this section, Kowk and Cheung (KWOK; CHEUNG, 2004) arranged the load balancing topic in four classes: (i) location policy; (ii) information policy; (iii) transfer policy and; (iv) selection policy. Observing the context of MigBSP, location policy treats the selection of a target resource for a candidate process for migration. The target resource choosing is performed by the Set Manager of the Set which is indicated in the $PM(i, j)$. This manager computes the function $time(p)$ in order to evaluate which processor finishes all the work assigned to it (amount of instructions of the BSP processes that it executes) in the least time. After this procedure, the verification of the migration viability occurred. This last test considers both the communication and migration costs issues.

Information policy determines which data are collected for load balancing execution. MigBSP analyses information from three source metrics: (i) Computation; (ii) Communication and; (iii) Memory. The selection of the first two is explained because a BSP

application is divided in supersteps, each one with a computation and communication phases. The computation part captures information about the amount of instruction executed at superstep as well as the processing time spent to compute it. The communication part treats data about the communicated bytes and the time spent in this phase during the superstep execution. On the other hand, data referred to memory are useful to measure the process transferring cost to continue its execution on another processor. In this context, we observed the the time to transfer the whole memory image of the process between the current and the target Set and the time to perform specific process migration operations.

Transfer policy leads with issues like the moments where the load transferring takes place, who takes the initiative to implement it and which resources enter as possible candidates to receive a process. Firstly, the BSP processes rescheduling launching is triggered when the interval $\alpha$ between supersteps is reached. This value is variable and depends on the environment stability (if processes are balanced or not) and if a pattern without migrations is observed on last rescheduling calls. MigBSP employs the sycnhronous load balancing detection approach (WATTS; TAYLOR, 1998), since all Set Managers know exactly the next moment for processes rescheduling and begin this procedure synchronously. Besides this, our model was designed to be implemented at middleware level. Concerning this, MigBSP does not impose modifications in application code proposing a migration know as objective (see Subsection 2.3 for details). Moreover, it is important to emphasize that the processes rescheduling attempting is not launched by either an overloaded or an idle processor. Finally, all processors that belong to the target Set are candidates to receive a process. They are evaluate and the most suitable among them is selected for migration viability testing.

Selection policy treats with issues like which processes are selected for migration. MigBSP chooses these candidate processes using its decision function called *PM* (Potential of Migration). This function was developed based on the notion of force from physics. *PM* is the resultant force of the following forces that act over the candidate process for migration: (i) Computation; (ii) Communication and; (iii) Memory. The greater the $PM(i, j)$, the larger the possibility to migrate process $i$ to Set $j$. Each process $i$ passes to its Set Manager its largest $PM(i, j)$ at the rescheduling moment. After that, the Set Managers exchange data among themselves. Lastly, each Set Manager has a list of the largest $PM(i, j)$ of each process and apply one of the two heuristics to select the candidate processes for migration: (i) the processes with *PM* larger than $Max(PM).x$ where $x$ is a percentage are selected and; (ii) Only the process with the highest *PM* is taken as candidate.

MigBSP uses a global strategy for load balancing (ZAKI; LI; PARTHASARATHY, 1997). In global schemes, the load balancing decision is made using a global knowledge, *i.e.*, data from all processes take part in the synchronization operation for processes replacement decisions. The list of the highest *PM* of all BSP processes is known by all Set Managers when attempting for migration. The main advantage of global schemes is that we can apply better load balancing decisions once data from all processes are considered. However, the synchronization is the most expensive part of this approach (ZAKI; LI; PARTHASARATHY, 1997). However, we take profit from BSP model organization, which already imposes a barrier synchronization among the processes. Therefore, we do not need to pay an addition cost to use the global approach which causes its selection for developing MigBSP.

Inside global classification (WARAICH, 2008), MigBSP presents a hybrid load balancing between centralized and distributed approaches. While the centralized one works

with a process whose is a master balancer, the distributed approach informs that the load balancer is replicated on all the processes. The profile information is broadcast to every other processor in this last technique. Differently from them, MigBSP employs the concept of Set Managers whose are responsible for load balancing computation. Thus, we do not need to perform a broadcast among all processes and we do not have a central point of control, which naturally presents scalability problems.

## 4.10 Summary

MigBSP answers the following issues about processes migration: (i) "When" to launch the migration; (ii) "Which" processes are candidates for migration and; (iii) "Where" to put an elected process. The decision for processes remapping is taken at the end of a superstep (after the barrier). This migration point was chosen because in this moment it is possible to analyze data from all BSP processes at their computation and communication phases. Aiming to generate the least intrusiveness in application as possible, we applied two adaptations that control the value of $\alpha$. $\alpha$ is updated at each rescheduling call and will indicate the interval to the next one. Aiming to store the variations on system state, a temporary variable called $\alpha'$ is used and updated at each superstep through the increment or decrement of one unit. $\alpha$ is filled with $\alpha'$ value in the moment of rescheduling call. The adaptations' ideas are: (i) to postpone the rescheduling call if the system is stable (processes are balanced) or to turn it more frequent, otherwise; (ii) to delay this call if a pattern without migrations on $\omega$ past calls for rescheduling is observed. Aiming to analyze the system stability, the times of the slowest and fastest processes at each superstep are captured as well as the average time is computed. A variable $D$ is used to indicate a percentage of how far the slowest and the fastest processes may be from the average to consider the processes as balanced. According to Shah et al. (SHAH; VEERAVALLI; MISRA, 2007), MigBSP should be classified as an adaptive scheduler whose parameters change based on the global state of the system (see Section 2.1 for details).

The answer for "Which" is solved through our decision function called Potential of Migration (PM). Each process $i$ computes $n$ functions $PM(i,j)$, where $n$ is the number of Sets and $j$ means a Set. The idea consists in performing a subset of the processes-resources tests at the rescheduling moment. $PM(i,j)$ is found using Computation, Communication and Memory metrics. The relation among them is based on the notion of force from physics. Computation and Communication act in favor of migration, while Memory works in an opposite direction. Computation metric - $Comp(i,j)$ - considers a Computation Pattern $P_{comp}(i)$ that measures the stability of a process $i$ regarding the amount of instructions at each superstep. This value is close to 1 if the process is regular and close to 0 otherwise. This metric also performs a computation time prediction based on all computation phases between two activations of processes rescheduling. For this prediction it is used the Aging concept (TANENBAUM, 2003). In the same way, Communication metric - $Comm(i,j)$ - computes the Communication Pattern $P_{comm}(i,j)$ between processes and Sets. Furthermore, this metric uses communication time prediction considering data between two rebalancing activations. Memory metric - $Mem(i,j)$ - considers process memory, transferring rate between considered process and the manager of target Set, as well as migration costs.

$PM(i,j)$ selects the candidate processes for resource relocation. $PM(i,j)$=$Comp(i,j)+Comm(i,j)-Mem(i,j)$. A high $PM(i,j)$ means that process $i$ has high computation time, high communication with processes that belong to Set $j$ and presents low migration costs.

BSP processes calculate $PM(i, j)$ locally. At each rescheduling call, each process passes its highest $PM(i, j)$ to its Set Manager. This last entity exchanges the $PM$ of its processes to other managers. There are two heuristics to choose the candidates, both based on a decreasing ordered list of $PMs$. The heuristics are: (i) processes that have $PM$ higher than a $MAX(PM).x$ are candidates, where $x$ is a percentage; (ii) choose just one process.

$PM(i, j)$ of a candidate process $i$ is associated to a Set $j$. The manager of this Set will select the most suitable processor to receive the process $i$. Before any migration, its viability is verified considering the following data: (i) the external load on source and destination processors; (ii) the BSP processes that both processors are executing; (iii) the simulation of considered process running on destination processor; (iv) the time of communication actions considering local and destination processors, and; (v) migration costs. Concerning this, we computed two times: $t1$ and $t2$. $t1$ means the local execution of process $i$, while $t2$ encompasses its execution on the other processor and includes the migration costs. For each candidate is calculated the migration gain and its transferring viability.

Since the processes scheduling problem is NP-hard, heuristics are sought to tackle the problem (TSE, 2009). Heuristics are intended to gain computational performance at the cost of accuracy or precision. Especially on scheduling topic, heuristic techniques allow us to obtain a satisfactory mapping in a reasonable time (MARTÍNEZ-GALLAR; ALMEIDA; GIMÉNEZ, 2007; TSENG; CHIN; WANG, 2009). Concerning this, MigBSP uses heuristics in two situations: (i) to compute the decision function $PM$ and; (ii) to select the candidate processes for migration. Our idea was to offer a competitive scheduling model in two ways. The former one considers the performance gains with processes relocation through good decisions on choosing the candidates processes for migration as well as the processors to receive them. The second way is responsible to perform the scheduling computation and communication actions in an acceptable time. This issue is pertinent mainly when migrations are inviable.

# 5 MIGBSP MODEL EVALUATION

This chapter presents the strategies that were used to evaluate and validate MigBSP. Moreover, it presents the MigBSP's results on executing relevant BSP applications. We will present the situation where the employment of our model achieves better performance than application execution simply. Furthermore, we will explain in details the situations in which MigBSP includes an overhead in the total time of the BSP application. The simulation technique was utilized to validate MigBSP. This technique facilitates the management of the nodes, the networks as well as the BSP processes properly. It also makes both the implementation of processes migration and the MigBSP execution with different parameters easier.

Chapter 5 is segmented in 7 sections organized as follows. Section 5.1 describes the main objectives of this chapter. Section 5.2 characterizes the requirements for simulating MigBSP. The descriptions of the evaluation methodology, the different execution scenarios and our multi-cluster topology are expressed in Section 5.3. Sections 5.4, 5.5 and 5.6 show the results when simulating three applications with MigBSP. These sections are the core of this chapter, since they will inform the MigBSP's behavior and its viability, or not, for obtaining better performance on BSP-based application executions. Finally, Section 5.7 shows the main topics written in this chapter and directs the reader to the Conclusion chapter.

## 5.1 Objectives

The main aim of this evaluation chapter is to demonstrate the viability of MigBSP to reschedule BSP processes. Therefore, our idea is to present that application execution should be faster with MigBSP. In addition, we must explain the situation where MigBSP does not collaborate to enhance application performance. An obvious approach for obtaining valid results is to conduct experiments on production platforms, or at least on large testbeds. Unfortunately, this often proves infeasible. Real-world platforms may not be available for the purpose of experiments, so as not to disrupt production usage. Results are often non-reproducible due to resource dynamics (*e.g.*, time-varying and non-deterministic network usage, unpredictable host failures). Furthermore, experiments on real-world platforms may be prohibitively time consuming especially if large numbers of experiments are needed to explore many scenarios with reasonable statistical significance.

Given all difficulties detailed in the last paragraph, the majority of published results are obtained in simulation (MARTÍNEZ-GALLAR; ALMEIDA; GIMÉNEZ, 2007; NGUYEN et al., 2008; GROEN; HARFST; ZWART, 2009). Thus, we agree that the best approach to test MigBSP algorithms and to facilitate results reproduction is the use of simulation. Concerning this, we studied in details the functioning of the BSP model as

well as a spectrum of applications that are based on it (MCCOLL, 1995; SKILLICORN; HILL; MCCOLL, 1997; LEE, 2004; LOW; LIU; SCHMIDT, 2007). This study resulted in the following applications that will be analyzed to validate MigBSP.

(i) Lattice Boltzmann application - This application is large used in computational fluid dynamics area. Moreover, this application was chosen because its algorithm may be easily adapted to a large serie of other computing areas.

(ii) Smith-Waterman application - This application is based on dynamic programming and it is characterized by variation in the computation intensity along the matrix cells.

(iii) LU decomposition application - This application presents an algorithm where a matrix is written as the product of a lower triangular matrix and an upper triangular matrix. The decomposition is used to solve systems of linear equations as well as to turn the calculation of the determinant of a matrix easier.

The evaluation of these three applications will point general situations where the migrations are profitable and where MigBSP execution just introduces more overhead on application execution. The analysis of the tested applications represents the main objective of MigBSP evaluation. While the first application is regular, the other two present an irregular behavior. The regularity issue treats the processes' activities at each superstep. Thus, the behaviors of the BSP processes on the last two applications change along the execution due to fluctuations on the number of instructions and/or on the amount of communicated bytes that the processes perform at each superstep.

## 5.2 Simulating MigBSP Model

We studied grid application simulators to develop MigBSP and the considered BSP applications. A simulator must present at least the following characteristics to complete our requirements.

- Processes creation - Each process must execute a specific region of code or a function.
- Message passing-based application development - The simulator must allow all-to-all message passing among the application processes. In addition, it is preferable that it offers asynchronous communication, since BSP model affirms the interaction among the processes must occur in this manner.
- Processes migration - A mechanism for process relocation to another resource must be available.
- Special directives - The simulator must present directives to capture the current simulation time, the number of instructions as well as the communicated bytes in message passing actions.

The last phase of a BSP superstep is the synchronization barrier. It is possible to implement it using simple send-receive communication directives. In addition, asynchronous communication can be designed using a synchronous communication directive,

a communication request queue and a special thread that handles the queue. After the description of the requirements, we verified different simulators organized as follows: (i) MicroGrid (LIU; CHIEN, 2004); (ii) Simgrid (CASANOVA; LEGRAND; QUINSON, 2008); (iii) Gridsim (SULISTIO et al., 2007); (iv) Optorsim(CAMERON et al., 2004). These four simulators offer the construction of a heterogeneous environment regarding the speed of both the communication networks and the processors.

MicroGrid[1] provides transparent virtualization of resources, networks and information services, allowing the direct study of complex applications. OptorSim[2] are specifically designed to study data replication on grids. GridSim [3] was initially intended for grid economy, even if it became used in other areas of grid computing. Simgrid[4] aims to facilitate research in the area of distributed and parallel application scheduling on distributed computing platforms. This last simulator was chosen as execution tool since it fills out our requirements and it is widely used by the scientific community[5].

We are using MSG interface from Simgrid. This interface allows the study of CSP-based (concurrent sequential processes) applications. MSG offers an API for managing functions of processes (agents), hosts and tasks. In addition, MSG provides functions for operating system management. The main functions of this last area treat the execution of a task, the process sleeping, time capturing as well as the sending and the receiving of tasks to/from other processes.

### 5.2.1 Platform and Processes Deployment Definition

Simgrid allows us to specify two kinds of files (MUNCK; VANMECHELEN; BROECKHOVE, 2009): (i) computing platform and; (ii) processes deployment. Both are written in XML format. The former contains the description of hosts, network routes and inter-hosts network connections. Concerning the host features, we can fill its power and availability. Network links represent one-hop network connections. They are characterized by their identification and their bandwidth. The latency is optional with a default value of 0.0. Figure 5.1 (a) depicts a platform example with two nodes (hosts). Simgrid assumes that each host has just one processor. The code to build this infrastructure is shown in Figure 5.2.

Figure 5.1 also illustrates the deployment of the processes. The code for this deployment is expressed in Figure 5.3. Firstly, each process must be mapped to a host. This procedure determines the first processes-processors assignment and must be provided by the programmer/user. MigBSP may change this mapping during application runtime. XML definition allows us to specify arguments for each process. Considering Figure 5.1 (a), the process whose executes on Node1 defines the receiver node as its argument.

The assembly of a cluster may occur through the definition of multiple hosts and routes. However, the last version of Simgrid simplifies the creation of large-scale clusters in few lines (using the tag called Set). The Set tag of Simgrid platform file requires the filling of the prefix, the suffix and the radical parameters. Thus, we can establish large-scale clusters in few lines. For each Set in Simgrid we can write the power of each host as well as the backbone network which connects its nodes. Figure 5.1 (b) illustrates a multi-cluster platform. Its development requires the specification of two clusters, a network link

[1] http://www-csag.ucsd.edu/projects/grid/microgrid.html
[2] http://edg-wp2.web.cern.ch/edg-wp2/optimization/optorsim.html
[3] http://www.gridbus.org/gridsim/
[4] http://simgrid.gforge.inria.fr/
[5] http://simgrid.gforge.inria.br/doc/people.html

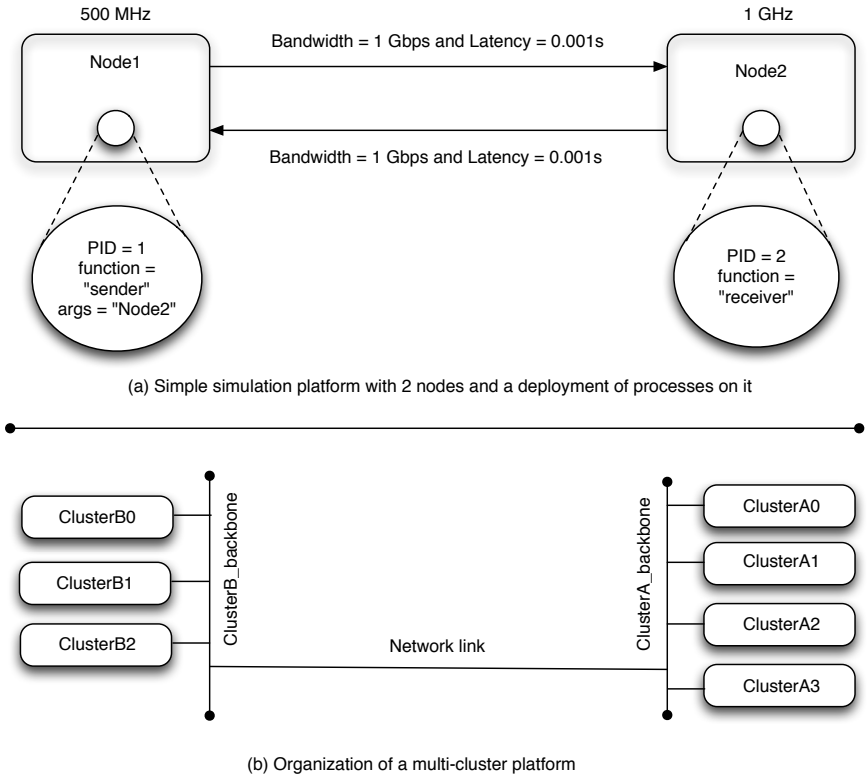


Figure 5.1: Simple execution infrastructure with two BSP processes

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">
  <host id="Node1" power="500000000"/>
  <host id="Node2" power="1000000000"/>
  <link id="link" bandwidth="1000000000" latency="0.001"/>
  <route src="Node1" dst="Node2">
    <link:ctn id="link"/>
  </route>
  <route src="Node2" dst="Node1">
    <link:ctn id="link"/>
  </route>
</platform>
```

Figure 5.2: Simgrid XML file for platform description

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">
  <process host="Node1" function="sender">
  <argument value="Node2"/>
  </process>
  <process host="Node2" function="receiver"/>
</platform>
```

Figure 5.3: Simgrid XML file for processes deployment

and a connection between the clusters.

### 5.2.2 Writing BSP Application

Simgrid application is written in C programming language. This application must implements the functions described in the deployment file. Both platform and deployment files are passed as parameters when executing a Simgrid application. Although the simulation creates a virtual distributed system, its execution is local just using one computer. We can write a Simgrid application according to either SPMD (Simple Program Multiple Data) or MPMD (Multiple Program Multiple Data) style. Each process should use the directive MSG_Process_self_PID() to indicate specific sub-regions of code to execute.

Simgrid relies on a concept of task that can be either computed locally or transferred to another processor in the simulated infrastructure. Simgrid developers agree that this abstraction is the right one for scheduling algorithms. A task may then be defined by a computing amount, a message size and some private data. Simgrid application must use this abstraction to express computation. Therefore, we must estimate the number of computing instructions from a code of a real application to create a task which will simulate the same (or approximate) computing time. MSG_task_execute() function executes a task and waits for its termination. MSG_get_clock() function is used to capture the current simulation time. Its usage before and after executing a task shows the time spent to compute it in the host which the process belongs to.

Simgrid offers synchronous message passing communication natively. MSG_task_put puts a task on a channel of a host and waits for the end of the transmission. MSG_task_get-_from() listens on channel and waits for receiving a task from host. Clock directive may be used to save the communication time. Simgrid does not provide the barrier and collective communication functions. Process migration is offered by Simgrid through function MSG_process_change_host(). This function returns automatically and does not impose any cost related to migration. We can use the function MSG_process_sleep() to obtain a delay on process execution, simulating the migration costs involved on its transferring. Finally, the simulation returns the total time (in seconds) for application execution. This time is different from the real observed one. For example, a simulated time of 234.34s can last just 4.00s of simulation in the real world.

### 5.2.3 MigBSP Development

We implemented the barrier mechanism using a simple centralized algorithm. The process with identification equal to 1 receives messages from all other processes. When it receives the last message, it signalizes them to start the computation of the next superstep. We opted for this algorithm for simplicity. Thus, we can focus our effort on developing MigBSP algorithms. Both executions with and without our model use the same barrier implementation. Each process captures its superstep time at the barrier function. They save this time in a local vector at each superstep and just pass this vector to their Set Manager when rescheduling is activated. This strategy is useful to minimize network interactions among BSP processes and their respective Set Managers. Barrier function is also used to update the Computation, Communication and Memory metrics. In addition, the control of $\alpha'$ and the call for processes scheduling attempting (when $\alpha$ is reached) are implemented at this moment.

We developed wrappers to save the computation and communication times. They act over the MSG_task_execute() and MSG_task_get_from() functions. In the beginning of each function we capture the number of instructions or bytes and take the initial clock. At the end of them, we take the clock again in order to evaluate the total time to execute

the procedure. MigBSP works with asynchronous communication on sending operations. Given that MigBSP does not offer asynchronous communications natively, we created an auxiliary thread and a request queue per processes for treating these operations. Auxiliary thread executes a loop which pulls the first element of the queue and proceeds with the requisition. The thread is killed after passing the last application superstep.

The deployment file firstly describes the processes that act as Set Managers. They execute the function manager() and receive as parameter a list which informs the nodes under its jurisdiction. The deployment follows with the sequence of BSP processes. The deployment file indicates the target node, a function called superstep() and the identification of the Set Manager when completing the data about a specific BSP processes. The Set Managers reach the processes varying their nodes under their responsibilities and verifying the BSP processes that are executing on them. Each process interacts with its Manager through the identification passed as parameters when it was initialized. The call for processes rescheduling returns one or more candidate processes, depending on the employed heuristic. The migration procedure is performed through a combination of the function MSG_process_change_host() execution followed by a delay equal to Memory metric $Mem(i, j)$ value considering a process $i$ and a Set $j$. Function MSG_process_sleep() is used to offer the delay facility.

## 5.3 Scientific Methodology

This section describes some decisions for evaluating MigBSP. Firstly, we present how we obtained an estimative of the migration costs to transfer a process in our infrastructure. Following this, we discuss in details the different scenarios for testing MigBSP. The key idea on this topic is to evaluate different execution situations to get conclusions about the MigBSP's behavior easier. Finally, we show the infrastructure testbed that was assembled to evaluate MigBSP as well as the initial processes-resources deployment on it.

### 5.3.1 Analyzing the Migration Costs

Aiming to work with migration costs close to the real situations, we implemented a simple application and tested it with AMPI communication library. Our results can be seen in details in (PILLA; ROSA RIGHI; NAVAUX, 2008). The key idea of the tests was to observe the migration costs of a process when varying the memory space allocated in it. We reserved three machines and created two processes in order to perform the tests. The first process allocates a specific amount of data and calls the directive AMPI_MigrateTo(destination). Besides this, it sends two messages to the second process, one before and another after migration. The second process was created on the machine that was not involved in the migration of the first process. It measure the time between the two messages through the directive MPI_Wtime().

The tests were executed in three Pentium III 1.2 GHz nodes of the Corisco cluster located at the Institute of Informatics from UFRGS. The nodes are connected through a 100 Mbit/s network. The number of bytes allocated by the first process varies from 50 bytes up to $10^7$ bytes. The application was tested 50 times for each amount of considered bytes and an arithmetic average was measured. Figure 5.4 depicts the results when executing the synthetic application. The results increase almost linearly when increasing the amount of process' data. We observed 1.21 seconds when migrating a process that allocated 10 MBytes of data. This time represents a rate close to 70% of the theoretical capacity of the network. The results indicate a viable way to use AMPI as tool to implement MigBSP in

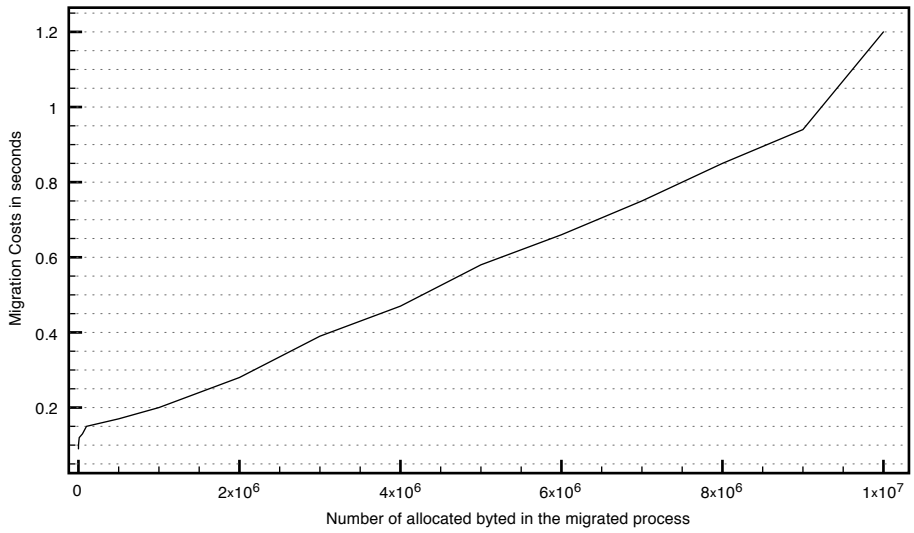


Figure 5.4: Migration time with AMPI when varying the allocated data in the transferred process

the future.

Our experimental tests measured the migrations costs in a single cluster since AMPI offers treatment for this kind of architecture. However, MigBSP works with the idea of an architecture composed by multiple sites. In this context, we developed a technique to measure an estimative of AMPI migration costs for our purposes. Firstly, we modeled the time to migrate a process ($mt_1$) presented on $y$ axis in Figure 5.4 as the process' size divided by the cluster bandwidth. Considering that the process' image remains the same, the migration time on a multi-cluster environment ($mt_2$) is measure as $\frac{mt_1 \,.\, bw_1}{bw_2}$. $bw_2$ means the average bandwidth from the source up to the destination processor in the multi-cluster environment. For instance, if we use 100Mbits/s and 0.2s as $bw_1$ and $mt_1$ to migrate a process with 1 Mbyte in memory, new $mt_2$ in a multi-hop network with mean bandwidth of 60Mbits/s will be 0.33s.

### 5.3.2 Scenarios of Tests

We modeled three scenarios to evaluation MigBSP: (i) Application execution simply; (ii) Application execution with MigBSP scheduler without applying migrations and; (iii) Application execution with MigBSP scheduler allowing migrations. Table 5.1 summarizes the observed scenarios. Our idea with this organization is to show the normal behavior of MigBSP (scenario iii) and the situation where all migrations are inviable (scenario ii). Scenario ii measures the overhead related to scheduling computation and the costs of message exchanging among Set Managers as well as message passing between BSP processes and their respective Set Managers. In other words, scenario ii consists in performing all scheduling calculus and all decisions about which processes will really migrate, but it does not comprise any migrations actually. Scenario iii enables migrations and adds the migrations costs to those processes that migrated from one processor to another.

The comparison between scenarios i and ii refers to the overhead imposed by MigBSP on application execution when migrations do not take place. Scenario ii always present times larger than scenario i. The difference between them represents exactly the overhead imposed by MigBSP. The analysis of scenarios i and iii will show the final gain or loss

Table 5.1: Different scenarios for evaluating MigBSP

| Tested scenarios | Application execution | MigBSP execution | Enabling Migrations |
|---|---|---|---|
| Scenario i | • | | |
| Scenario ii | • | • | |
| Scenario iii | • | • | • |

of performance when processes migration is applied. If the application execution time is reduced when using migrations, we can affirm that the relocation of one or more BSP processes outperforms the costs related to both the MigBSP scheduling and the transferring of migrated processes. The verification of both scenarios ii and iii is pertinent to observe the modification in time from a situation that we just have the overhead of MigBSP scheduling to another which migrations may occur.

### 5.3.3 Multi-Cluster Testbed Architecture

As grid technologies gain in popularity, separate clusters are increasingly being interconnected to create multi-cluster computing architectures for the processing of scientific and commercial applications (ZHANG; KOELBEL; COOPER, 2009). Considering this, we modeled an infrastructure with 5 clusters as illustrated in Figure 5.5. All processors have the same machine architecture. Each cluster is mapped to a Set which presents the execution of the Set Manager on its first node. Our infrastructure is heterogeneous in terms of processing capacity of the nodes as well as in terms of bandwidth of the network links. In addition, we do not apply variations of performance on the nodes nor on the network links.

Any node inside our parallel architecture offers just one processor for executing BSP processes. Figure 5.6 illustrates the initial processes-processors mapping for our tests. The basic idea is to fill one cluster and then to pass to another one. We map one process per node owing to each one has a single processor. If the amount of process is greater than processors, the mapping begins again from the first Set. This happens when 200 processes must be mapped on our infrastructure. The next sections will discuss the results of MigBSP when executing different BSP applications. We are using the notation {(px, ny)} for denoting a process px that is executing or was reassigned to run in the node ny.

## 5.4 Lattice Boltzmann Method

The Lattice Boltzmann method is a powerful technique for the computational modeling of a wide variety of complex fluid flow problems (SCHEPKE CLAUDIO; MAILLARD, 2007). It is a discrete computational method based upon the Boltzmann equation. It considers a typical volume element of fluid to be composed of a collection of particles that are represented by a particle velocity distribution function for each fluid component at each grid point.

### 5.4.1 Modeling

We modeled a BSP implementation of a 2D-based Lattice Boltzmann Method to Simgrid using vertical domain decomposition (ROSA RIGHI et al., 2009b). The data volume is divided into spatially contiguous blocks along one axis. Multiple copies of the same program run simultaneously, each operating on its own block of data. Each copy of the

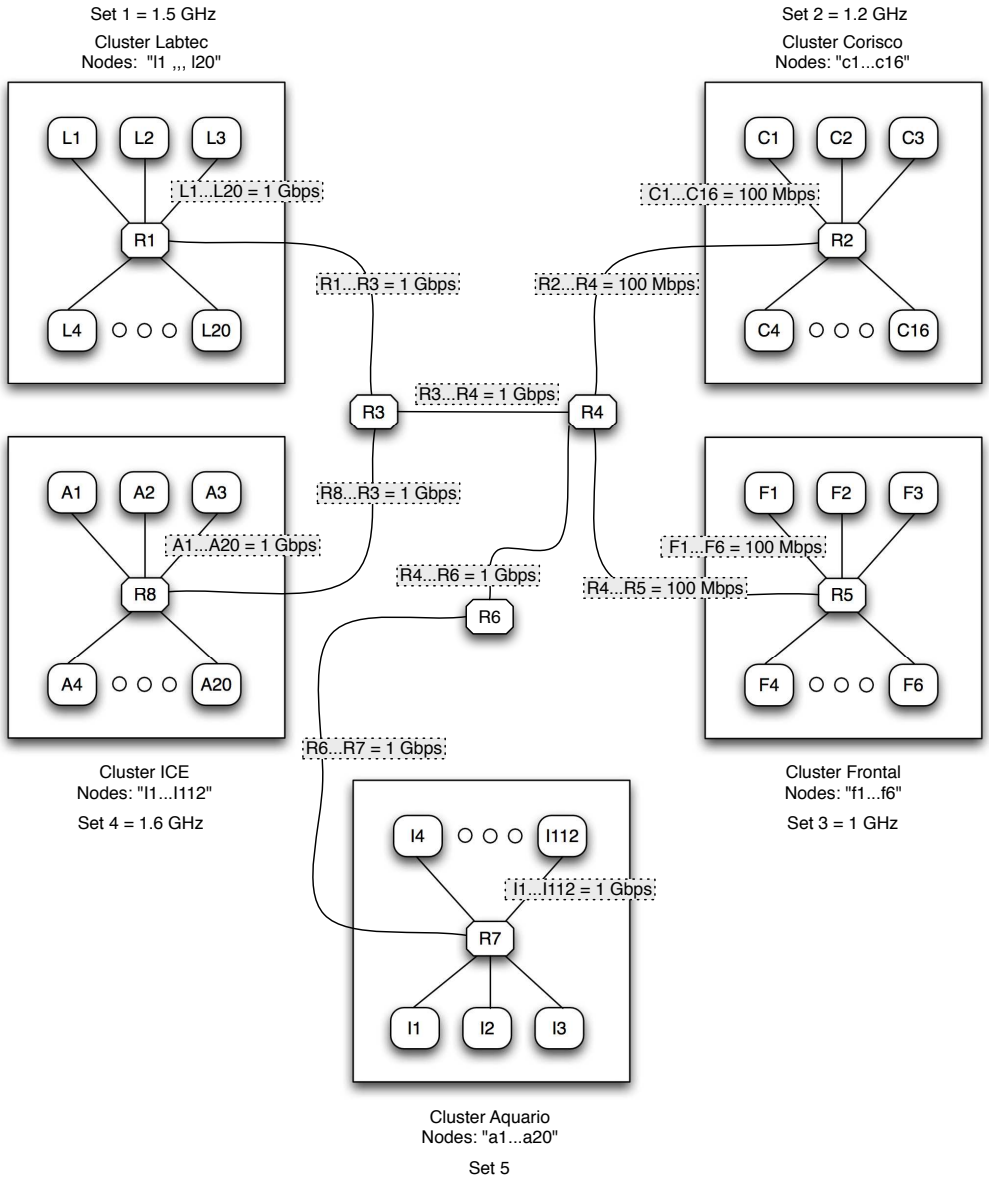


Figure 5.5: Heterogeneous testbed infrastructure with 5 Sets

| Initial Processes-Resources Mapping | |
|---|---|
| 10 processes | = L {1-10} |
| 25 processes | = L {1-20}, C {1-5} |
| 50 processes | = L {1-20}, C {1-16}, F {1-6}, I {1-8} |
| 100 processes | = L {1-20}, C {1-16}, F {1-6}, I {1-58} |
| 200 processes | = L {1-20}, C {1-16}, F {1-6}, I {1-112}, A {1-20}, L {1-20}, C {1-6} |

Figure 5.6: Initial processes-resources mapping

program runs as an independent BSP process. At the end of each iteration, data for the planes that lie on the boundaries between blocks are passed between the appropriate processes and the superstep is completed. An abstract view of the problem may see in Figure 5.7.

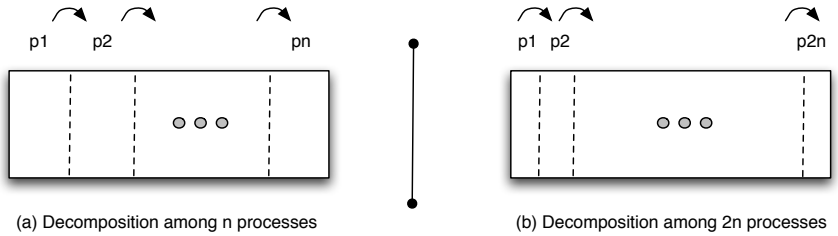


Figure 5.7: Different matrix partition organizations when varying the number of used processes

Besides Lattice Boltzmann, the developed scheme encompasses a broad spectrum of scientific computations, from mesh based solvers, signal processing to image processing algorithms. The considered matrix requires the computation of $10^{10}$ instructions and occupies 10 Megabytes in memory. These values were adopted based on real executions of Lattice Boltzmann method in our clusters at UFRGS, Brazil. As we can observe in Figure 5.7, matrix partition will influence the number of instructions to be executed per process and, consequently, its size in memory. Nevertheless, the quantity of communication remains the same independent of the used partition scheme. It is important to emphasize that our modeling may be characterized as regular, where each superstep presents the same number of instructions to be computed by processes as well as the same communication behavior.

When using 10 processes, each one is responsible for a sub-lattice computation of $10^9$ instructions, occupies 1.5 Megabyte (500 Kbytes is fixed to other process' data) and passes 100 Kilobytes of boundary data to its right neighbor. In the same way, when 25 processes are employed, each one computes $4.10^8$ instructions and occupies 900 Kbytes in memory. Finally, initial tests were executed using $\alpha$ equal to 4, 8 and 16. Furthermore, we employed $\omega$ equal to 3, 0.5 for initial $D$ and used heuristic two to choose the candidate process for migration. This heuristic affirms that at most 1 process will migrate at each rescheduling call.

### 5.4.2 Results

Table 5.2 presents the times when executing 10 processes. Firstly, we can observe that MigBSP's intrusivity on application execution is short when comparing both scenarios i and ii (overhead lower than 5%). The processes are balanced among themselves with this configuration, causing the increasing of $\alpha$ at each call for processes rescheduling. This explain the low impact when comparing scenarios i and ii. Besides this, MigBSP decides that migrations are inviable for any moment, independing on the amount of executed supersteps. In this case, our model causes a loss of performance in application execution. We always obtained negative values of *PM* when processes rescheduling was tested. This fact resulted in an empty list of candidate processes for migration.

The results of the execution of 25 processes are presented in Table 5.3. In this context, the system remains stable and $\alpha$ grows at each rescheduling call. One migration occurred {(p21,a1)} when testing 10 supersteps and using $\alpha$ equal to 4. Our notation informs that process p21 was reassigned to run on node a1[6]. A second and a third migrations happened when considering 50 supersteps: {(p22,a2), (p23,a3)}. They happened in the next

[6]Processes organization as well as the first processes-resources mapping can be seen in Figures 5.5 and 5.6

Table 5.2: Evaluating 10 processes on three considered scenarios (time in seconds)

| Super-step | Scenario i | $\alpha = 4$ | | $\alpha = 8$ | | $\alpha = 16$ | |
|---|---|---|---|---|---|---|---|
| | | Scen. ii | Scen. iii | Scen. ii | Scen. iii | Scen. ii | Scen. iii |
| 10 | 6.70 | 7.05 | 7.05 | 7.05 | 7.05 | 6.70 | 6.70 |
| 50 | 33.60 | 34.59 | 34.59 | 34.26 | 34.26 | 34.04 | 34.04 |
| 100 | 67.20 | 68.53 | 68.53 | 68.20 | 68.20 | 67.87 | 67.87 |
| 500 | 336.02 | 338.02 | 338.02 | 337.69 | 337.69 | 337.32 | 337.32 |
| 1000 | 672.04 | 674.39 | 674.39 | 674.06 | 674.06 | 673.73 | 673.73 |
| 2000 | 1344.09 | 1347.88 | 1347.88 | 1346.67 | 1346.67 | 1344.91 | 1344.91 |

Table 5.3: Evaluating 25 processes on three considered scenarios (time in seconds)

| Super-steps | Scenario i | $\alpha = 4$ | | $\alpha = 8$ | | $\alpha = 16$ | |
|---|---|---|---|---|---|---|---|
| | | Scen. ii | Scen. iii | Scen. ii | Scen. iii | Scen. ii | Scen. iii |
| 10 | 3.49 | 4.18 | 4.42 | 4.42 | 4.44 | 3.49 | 3.49 |
| 50 | 17.35 | 19.32 | 20.45 | 18.66 | 19.44 | 18.66 | 19.42 |
| 100 | 34.70 | 37.33 | 38.91 | 36.67 | 37.90 | 36.01 | 36.88 |
| 500 | 173.53 | 177.46 | 154.87 | 176.80 | 161.48 | 176.80 | 179.24 |
| 1000 | 347.06 | 351.64 | 297.13 | 350.97 | 303.72 | 350.31 | 317.96 |
| 2000 | 694.12 | 699.47 | 592.26 | 698.68 | 599,14 | 697.43 | 613.88 |

Table 5.4: Evaluating 50 processes on three considered scenarios (time in seconds)

| Super-steps | Scenario i | $\alpha = 4$ | | $\alpha = 8$ | | $\alpha = 16$ | |
|---|---|---|---|---|---|---|---|
| | | Scen. ii | Scen. iii | Scen. ii | Scen. iii | Scen. ii | Scen. iii |
| 10 | 2.16 | 2.95 | 3.20 | 2.95 | 3.17 | 2.16 | 2.16 |
| 50 | 10.78 | 13.14 | 14.47 | 12.35 | 13.32 | 12.35 | 13.03 |
| 100 | 21.55 | 24.70 | 26.68 | 29.91 | 25.92 | 23.13 | 24.63 |
| 500 | 107.74 | 112.46 | 106.90 | 111.67 | 115.73 | 111.67 | 117.84 |
| 1000 | 215.48 | 220.98 | 199.83 | 220.19 | 207.78 | 219.40 | 226.43 |
| 2000 | 430.95 | 436.79 | 408.25 | 435.88 | 417.56 | 434.68 | 434.30 |

Table 5.5: Evaluating 100 processes on three considered scenarios (time in seconds)

| Super-steps | Scenario i | $\alpha = 4$ | | $\alpha = 8$ | | $\alpha = 16$ | |
|---|---|---|---|---|---|---|---|
| | | Scen. ii | Scen. iii | Scen. ii | Scen. iii | Scen. ii | Scen. iii |
| 10 | 1.22 | 2.08 | 2.24 | 2.08 | 2.21 | 1.22 | 1.22 |
| 50 | 5.94 | 8.59 | 9.63 | 7.71 | 8.48 | 7.71 | 8.19 |
| 100 | 11.86 | 15.40 | 16.99 | 14.52 | 16.24 | 13.63 | 14.94 |
| 500 | 59.25 | 64.57 | 62.55 | 63.68 | 67.25 | 63.68 | 69.37 |
| 1000 | 118.48 | 124.69 | 113.87 | 123.80 | 119.06 | 122.92 | 129.46 |
| 2000 | 236.96 | 243.70 | 224.48 | 241.12 | 232.87 | 239.23 | 241.52 |

two calls for processes rescheduling (at supersteps 12 and 28). When evaluating 2000 supersteps and maintaining this value of $\alpha$, eight migrations take place: {(p21,a1), (p22,a2), (p23,a3), (p24,a4), (p25,a5), (p18,a6), (p19,a7), (p20,a8)}. We analyzed that all migrations occurred to the fastest cluster (Aquario). In addition, the first five migrations moved processes from cluster Corisco to Aquario. After that, three processes from Labtec cluster

Table 5.6: Evaluating 200 processes on three considered scenarios (time in seconds)

| Super-steps | Scenario i | α = 4 | | α = 8 | | α = 16 | |
|---|---|---|---|---|---|---|---|
| | | Scen. ii | Scen. iii | Scen. ii | Scen. iii | Scen. ii | Scen. iii |
| 10 | 1.04 | 2.86 | 3.06 | 1.95 | 2.11 | 1.04 | 1.04 |
| 50 | 5.09 | 10.56 | 17.14 | 9.65 | 11.06 | 7.82 | 8.15 |
| 100 | 10.15 | 16.53 | 25.43 | 15.62 | 21.97 | 14.71 | 16.04 |
| 500 | 50.66 | 57.84 | 68.44 | 56.93 | 71.42 | 55.92 | 77.05 |
| 1000 | 101.29 | 108.78 | 102.59 | 107.84 | 106.89 | 105.25 | 117.57 |
| 2000 | 200.43 | 209.46 | 194.87 | 208.13 | 202.22 | 204.69 | 211.69 |

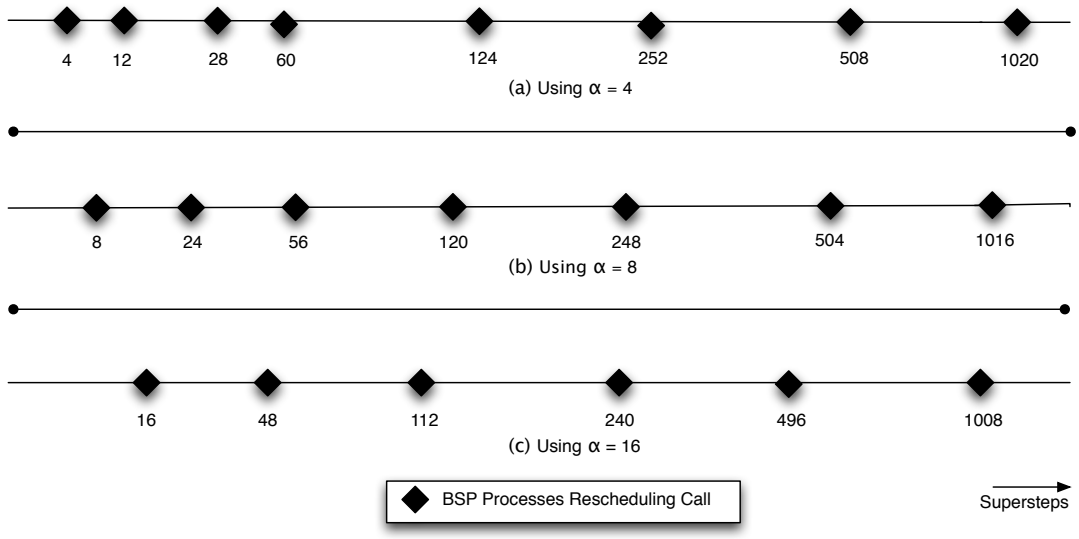


Figure 5.8: Amount of rescheduling calls with α 4, 8 and 16 when 25 processes and 2000 supersteps are evaluated

were chosen for migration. Concluding, we obtained a profit of 14% after executing 2000 supersteps in comparison of scenarios i and iii when α equal to 4 is used.

Analyzing scenario iii with α equal to 16, we detected that the first migration is postponed, which results in a larger final time when compared with lower values of α. With α 4 for instance, we have more calls for processes rescheduling with migrations during the first supersteps. This fact will cause a large overhead to be paid during this period. These penalty costs are amortized when the amount of executed supersteps increases. Thus, the configuration with α 4 outperforms other studied values of α when 2000 supersteps are evaluated. Figure 5.8 illustrates the frequency of processes rescheduling calls when testing 25 processes and 2000 supersteps. We can observe that 6 calls are done with α 16, while 8 are performed when initial α changes to 4. Considering scenarios ii, we conclude that the greater is α, the lower is the model's impact if migrations are not applied (situation in which migration viability is false).

Table 5.4 shows the results when the number of processes is increased to 50. The processes are considered balanced and α increases at each rescheduling call. In this manner, we have the same configuration of calls when testing 25 processes (see Figure 5.8). We achieved 8 migrations when 2000 supersteps are evaluated: {(p38,a1), (p40,a2), (p42, a3), (p39, a4), (p41, a5), (p37, a6), (p22, a7), (p21, a8)}. MigBSP moves all processes from cluster Frontal to Aquario and transfers two process from Corisco to the fastest cluster.

Using α 4, 430.95s and 408.25s were obtained for scenarios i and iii, respectively. Besides this 5% of gain with α 4, we also achieve a gain when α is equal to 8. However, the final result when changing initial α to 16 in scenario iii is worse than scenario i, since the migrations are delayed and more supersteps are need to achieve a gain in this situation.

Table 5.5 presents the execution of 100 processes over the tested infrastructure. As the situations with 25 and 50 processes, the environment when 100 processes are evaluated is stable and the processes are balanced among the resources. Thus, α increases at each call for processes rescheduling. The same migrations occurred when testing 50 and 100 processes, since the configuration with 100 just uses more nodes from cluster ICE. In general, the same percentage of gain was achieve with this amount of processes if compared with the execution of 50 processes.

The results of scenarios i, ii and iii with 200 processes is shown in Table 5.6. We have an unstable scenario in this situation, which explains the fact of a large overhead in scenario ii. Considering this scenario, α will begin to grow after ω calls for processes rescheduling without migrations. Taking into account scenario iii and α equal to 4, 2 migrations are done when executing 10 supersteps: {(p195,a1), (p197,a2)}. Besides these, 10 migrations take place when 50 supersteps were tested: {(p196,a3), (p198,a4), (p199,a5), (p200,a6), (p38,a7), (p39,a8), (p37,a9), (p40,a10), (p41,a11), (p42, a12)}. Despite the happening of these migrations, the processes are still unbalanced with adopted value of $D$ and, then, α does not increase at each superstep. After these migrations, MigBSP does not indicate the viability of other ones. Thus, after ω calls without migrations, MigBSP enlarges the value of $D$ and α begins to increase following adaptation 2 (see Section 4.5 for details).

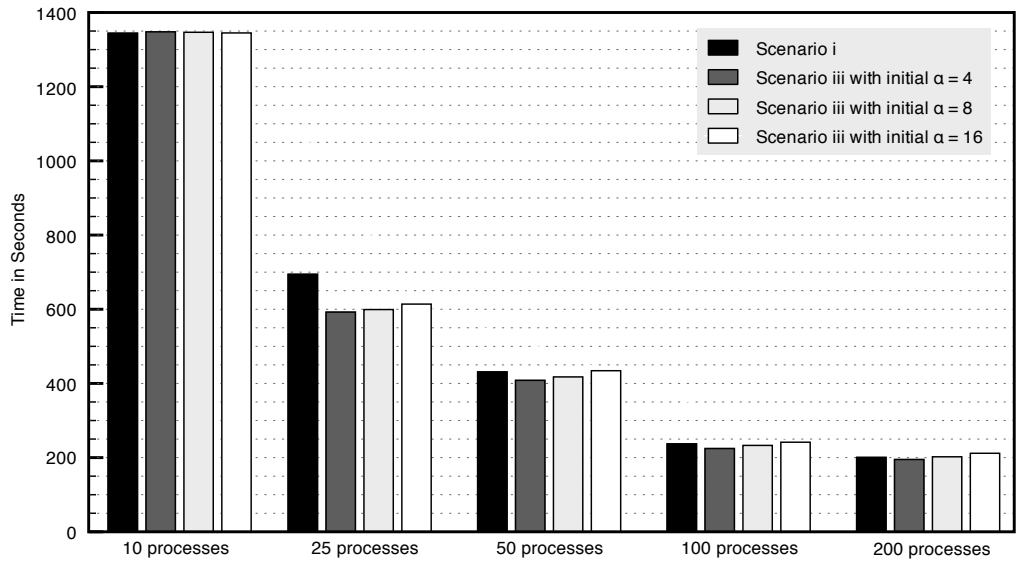


Figure 5.9: Analyzing the gain with processes migration considering both scenarios i and iii and 2000 supersteps

The graph in Figure 5.9 depicts the MigBSP's performance when migrations and 2000 supersteps are considered. Firstly, we observed that the greater is the number of processes, the better is the achieved results in scenario i. On the other hand, it is clear that this gain tend to stabilize up to a situation where a larger number of processes does not imply in a better performance. We verified the following issues about migrations. MigBSP obtained better results when 25 processes were tested. As we increase the number of

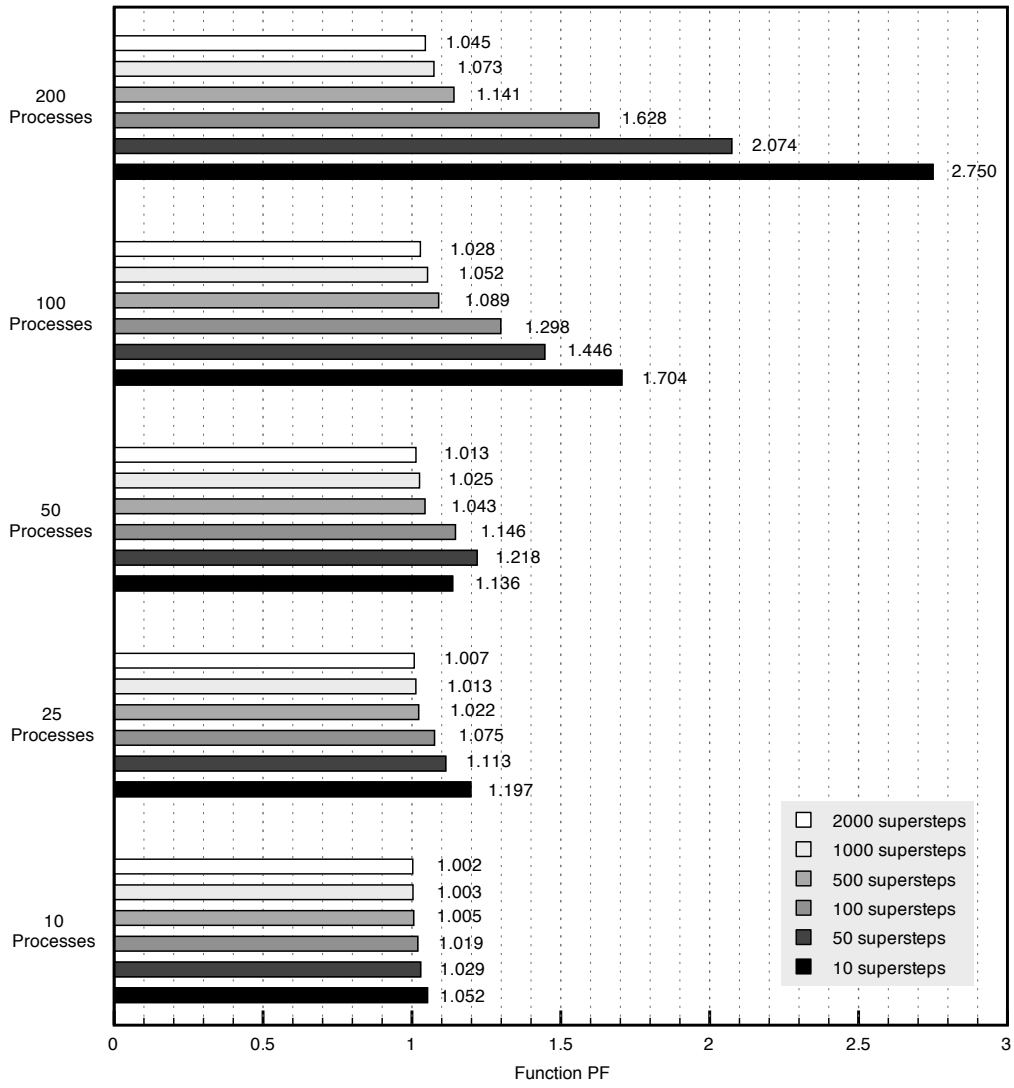


Figure 5.10: Observing MigBSP overhead when executing Lattice Boltzmann method with $\alpha$ 4. Performance Function (PF) on $x$ axis means $\frac{time\ in\ scenario\ ii}{time\ in\ scenario\ i}$

BSP processes, the considered amount of computation assigned to each one decreases as well. In addition, the greater is the number of processes, the higher is the flow of communications in the system. Therefore, we need more supersteps to obtain a gain with migrations when we enlarge the amount of evaluated processes.

Figure 5.10 shows the overhead of MigBSP when it uses 4 as initial $\alpha$. The main observation from it is the conclusion that we pay a large overhead when testing few supersteps. This is explained by the fact that MigBSP indicates migrations in the first supersteps and, consequently, a high cost is spent on such operations. The shorter is the number of supersteps, the shorter is the execution time to overlap the costs related to migrations. The evaluation of 10 supersteps is an instance of this assumption. The performance of scenario ii with 200 processes and 10 supersteps is larger then twice the time of scenario i. On the other hand, the simulation of 2000 supersteps comprises a scenario where the costs initially paid are amortized by a large execution with processes replacement. In this context, MigBSP represents a cost lower than 5% when executing 2000 supersteps.

Table 5.7 presents the barrier times captured when 2000 supersteps were tested. More

Table 5.7: Barrier times on two situations

| Processes | Scenario i - Without processes migration | Scenario iii - With processes migration |
|---|---|---|
| 10 | 0.005380s | 0.005380s |
| 25 | 0.023943s | 0.010765s |
| 50 | 0.033487s | 0.025360s |
| 100 | 0.036126s | 0.028337s |
| 200 | 0.043247s | 0.031440s |

especially, the time is captured when the last superstep is executed. We implemented a centralized master-slave approach for barrier, where process 1 receives and sends a scheduling message from/to other BSP processes. Thus, the barrier time is captured on process 1. The times shown in the third column of Table 5.7 do not include both scheduling messages and computation. Our idea is to demonstrate that the remapping of processes decreases the time to compute the BSP supersteps. Therefore, process 1 can reduce the waiting time for barrier computation since the processes reach this moment faster. Analyzing such table, we observed that a gain of 22% in time was achieved when comparing barrier time on scenarios i and iii with 50 processes. The gain was reduced when 100 processes were tested. This occurs because we just include more nodes from cluster ICE with 100 processes if compared with the situation where 50 are considered.

## 5.5 Smith-Watermann Application

Our second tested application is based on dynamic programming (DP), which is a popular algorithm design technique for optimization problems (LOW; LIU; SCHMIDT, 2007). Dynamic programming is a method for solving problems exhibiting the properties of overlapping subproblems and optimal substructure (LOW; LIU; SCHMIDT, 2007). DP algorithms can be classified according to the matrix size and the dependency relationship of each matrix cell. An algorithm for a problem of size $n$ is called a $tD/eD$ algorithm if its matrix size is O($n^t$) and each matrix cell depends on O($n^e$) other cells. In this thesis we concentrate on the parallelization of DP algorithms of the type $2D/1D$. These $2D/1D$ DP algorithms are all irregular with load computation density changes along the matrix's cells. In particular, we observed the functioning of Smith-Waterman algorithm (SMITH, 1988) that is a well-known 2D/1D algorithm for performing local sequence alignment.

### 5.5.1 Modeling

Smith-Waterman algorithm proceeds in a series of wavefronts diagonally across the matrix. Figure 5.11 (a) illustrates the concept of the algorithm for a $4\times4$ matrix with a column-based processes allocation. The more intense the shading, the greater is the load computation density of the cell. Each wavefront corresponds to a BSP superstep. For instance, Figure 5.11 (b) shows a $4\times4$ matrix that presents 7 supersteps. The computation load is uniform inside a particular superstep, growing up when the number of the superstep increases. Both organizations of diagonal-based supersteps mapping and column-based processes mapping bring the following conclusions (ROSA RIGHI et al., 2009a): (i) $2n-1$ supersteps are crossed to compute a square matrix with order $n$ and; (ii) each process will be involved on $n$ supersteps.

Each BSP process computes a block of data and sends it to other process at every

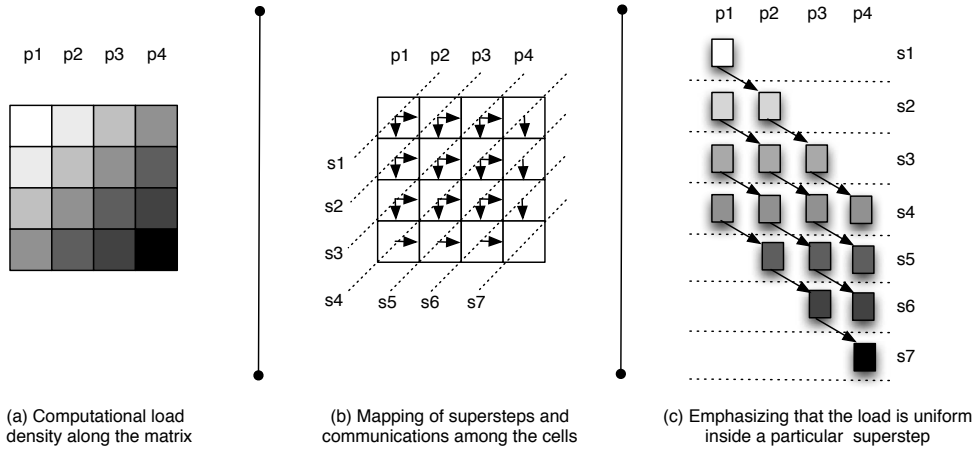


Figure 5.11: Different views of Smith-Waterman irregular application

superstep. Figures 5.11 (b) and (c) show the communication actions among the processes. Considering that cell $x,y$ ($x$ means a matrix' line, while $y$ is a matrix' column) needs data from the $x,y-1$ and $x-1,y$ other ones, we will have an interaction from process $py$ to process $py+1$. We do not have communication inside the same matrix column, since it corresponds to the same process.

The configuration of scenarios ii and iii depends on the Computation Pattern $P_{comp}(i)$ of each process $i$. $P_{comp}(i)$ increases or decreases depending on the prediction of the amount of performed instructions at each superstep. We consider a specific process as regular if the forecast is within a $\delta$ margin of fluctuation from the amount of instructions performed actually. In our experiments, we are using $10^6$ as the amount of instructions for the first superstep and $10^9$ for the last one. The increase of load computational density among the supersteps is uniform. In other words, we take the difference between $10^9$ and $10^6$ and divide by the number of involved supersteps in a specific execution. Considering this, we applied $\delta$ equal to 0.01 (1%) and 0.50 (50%) to scenarios ii and iii, respectively. This last value was used because $I_2(1)$ is $565.10^5$ and $PI_2(1)$ is $287.10^5$ when a $10\times10$ matrix is tested (see details about the notations in Subsection 4.6.1). The percentage of 50% enforces instruction regularity in the system. Both values of $\delta$ will influence the Computation metric, and consequently the choosing of candidates for migration. Scenario ii tends to obtain negatives values for *PM* since the Computation Metric will be close to 0. Consequently, no migrations will happen on this scenario.

We tested the behavior of square matrixes of order 10, 25, 50, 100 and 200. Each cell of a $10\times10$ matrix needs to communicate 500 Kbytes and each process occupies 1.2 Mbyte in memory (700 Kbytes comprise other application data). The cell of $25\times25$ matrix communicates 200 Kbytes and each process occupies 900 Kbytes in memory and so on. Initial tests were executed using $\alpha$ equal to 2, 4, 8 and 16. Furthermore, we employed $\omega$ equal to 3, initial $D$ equal to 0.5 and used heuristic one to choose the candidates for migration with $x$ equal to 80%. This heuristic takes data from the list that gathers the highest *PM* of each process and selects a percentage of processes as candidates.

### 5.5.2 Results

Table 5.8 presents the execution evaluation of BSP-based Smith-Waterman application. MigBSP executes 19 supersteps when crossing a $10\times10$ matrix. Adopting this size of matrix and $\alpha$ 2, 13.34s and 14.15s were obtained for scenarios i and ii which represents

a cost of 8%. The higher is the value of α, the lower is the MigBSP overhead on application execution. This occurs because the system is stable (processes are balanced) and α always increases at each rescheduling call. Three calls for processes relocation were done when testing α 2 (at supersteps 2, 6 and 14). The rescheduling call at superstep 2 does not produce migrations. At this step, the load computational density is not enough to overlap the consider migration costs involved on process transferring operation. The same occurred on the next call at superstep 6. The last call happened at superstep 14, which resulted on 6 migrations: {(p5,a1), (p6,a2), (p7,a3), (p8,a4), (p9,a5), (p10,a6)}. MigBSP indicated the migration of processes that are responsible to compute the final supersteps.

The execution with α equal to 4 implies in a shorter overhead since two calls were done (at supersteps 4 and 12). Observing scenario iii when testing a 10×10 matrix, we do not have migrations in the first call, but eight occurred in the other one. Processes 3 up to 10 migrated in this last call to cluster Aquario. α 4 outperforms α 2 for two reasons: (i) it does lesser rescheduling calls and; (ii) the call that causes processes migration was done at a specific superstep in which MigBSP takes better decisions. α 16 produces a migration of the last four processes after superstep 16. Migrations close to the final supersteps can be viewed as a good strategy, since the last processes are responsible for the largest computational loads. However, the remaining supersteps may not be enough to take profit from the migrations if the call is much closer to the end of the application.

Table 5.8: Evaluation of scenarios i, ii and iii when varying the matrix size

| Scenarios | | Matrix size | | | | |
|---|---|---|---|---|---|---|
| | | 10×10 | 25×25 | 50×50 | 100×100 | 200×200 |
| Scenario i | | 13.34s | 40.74s | 92.59s | 162.66s | 389.91s |
| Scen. ii | α = 2 | 14.15s | 43.05s | 95.70s | 166.57s | 394.68s |
| | α = 4 | 14.71s | 42.24s | 94.84s | 165.66s | 393.75s |
| | α = 8 | 13.78s | 41.63s | 94.03s | 164.80s | 392.85s |
| | α = 16 | 13.42s | 41.28s | 93.36s | 164.04s | 392.01s |
| Scen. iii | α = 2 | 13.09s | 35.97s | 85.95s | 150.57 | 374.62s |
| | α = 4 | 11.94s | 34.82s | 84.65s | 148.89s | 375.53s |
| | α = 8 | 13.82s | 41.64s | 83.00s | 146.55s | 374.38s |
| | α = 16 | 12.40s | 40.64s | 85.21s | 162.49s | 374.40s |

The system stays stable when the 25×25 matrix was tested. α 2 produces a gain of 11% in performance when considering 25×25 matrix and scenario iii. This configuration presents four calls for processes rescheduling, where two of them produce migrations. No migrations are indicated at supersteps 2 and 6. Nevertheless, processes 1 up to 12 are migrated at superstep 14 while processes 21 up to 25 are transferred at superstep 30. These transferring operations occurred to the fastest cluster. In this last call, the remaining execution presents 19 supersteps (from 31 to 49) to amortize the migration costs and to get better performance. The execution when using α 8 and scenario iii brings an overhead if compared with scenario i. Two calls for migrations were done, at supersteps 8 and 24. The first call causes the migration of just one process (number 1) to a1 and the second one produces three migrations: {(p21,a2),(p22,a3),(p23,a4)}. We observed that processes p24 and p25 stayed on cluster Corisco. Despite performed migrations, these two processes compromise superstep that includes them. Both are executing on slower cluster and barrier synchronization always waits for the slowest process.

Maintaining the 25×25 matrix size and adopting α 16, we have two calls for migration: at supersteps 16 and 48. This last call migrates p24 and p25 to cluster Aquario. Although this movement is pertinent to get performance, just one superstep is executed before finalizing the application. MigBSP does not have knowledge about application behavior and migrations are indicated even when the application is close to finish. Therefore, the migrations at superstep 48 contributed to increase overhead instead gain in application performance.

50 processes and 99 supersteps were evaluated when the 50×50 matrix was considered. In this context, α also increases at each call for processes rescheduling. We observed that an overhead of 3% was found when scenario i and ii were compared (using α 2). In addition, we observed that all values of α achieved a gain of performance in scenario iii. Especially when α 2 was used, five calls for processes rescheduling were done (at supersteps 2, 6, 14, 30 and 62). Migrations were not indicated in the first three calls. The greater is the matrix size, the greater is the number of supersteps needed to make migrations viable. This happens because our total load is fixed (independent of the matrix size) but the load partition increases uniformly along the supersteps (see Section 5.5.1 for details). Process 21 up to 29 are migrated to cluster Aquario at superstep 30, while process 37 up to 42 are migrated to this cluster at superstep 62. Using α equal to 4, 84.65s were obtained for scenario iii which results a gain of 9%. This gain is greater than that achieved with α 2 because now the last call for processes rescheduling is done at superstep 60. The same processes were migrated at this point. However, there are two more supersteps to execute with new processes location using α equal to 4.

Three calls for processes rescheduling were done with α8 (at supersteps 8, 24 and 56). Just the last two produce migration. Three processes are migrated at superstep 24: {(p21,a1),(p22,a2),(p23,a3)}. Process 37 up to 42 are migrated to cluster Aquario at superstep 56. This last call is efficient since it transfers all processes from cluster Frontal to Aquario. Two calls for processes relocation were done with α 16. At superstep 16, process 1 up to 15 are migrated to cluster Aquario. MigBSP selected as candidates for migration at superstep 48 the processes from p37 to p42. However, just five migrations occurred because cluster Aquario presents only 20 processors and p37 remains in the same previous location.

The execution with the 100×100 matrix shows good results with processes replacement. Six rescheduling calls were done when using α 2. At the first three (supersteps 2, 6 and 14) no migrations occur. Process 21 up to 29 are migrated to cluster Aquario after crossing superstep 30. In addition, process 37 to 42 are migrated to cluster Aquario at superstep 62. Finally, MigBSP indicates 7 migrations at superstep 126, but just 5 occurred: p30 up to p36 to cluster Aquario. These migrations complete one process per node on cluster Aquario. MigBSP selected for migration those processes that belonged to cluster Corisco and Frontal, which are the slower clusters on our infrastructure testbed. α equal to 16 produced 3 attempts for migration when 100×100 matrix is evaluated (at supersteps 16, 48 and 112). All of them triggered migrations. In the first call, the 11 first processes are migrated to cluster Aquario. All process from cluster Frontal are migrated to Aquario at superstep 48. Finally, 15 processes are selected as candidates for migration when the application finishes the computation of 112 supersteps. They are: p21 to p36. This spectrum of candidates is equal to the processes that are running on Corisco. Considering this, only 3 processes were migrated actually: {(p34,a18),(p35a19),(p36,a20)}.

Table 5.8 also shows the application performance when the 200×200 matrix was tested. The overhead of our model is smaller than 2% for scenario ii. Furthermore, satis-

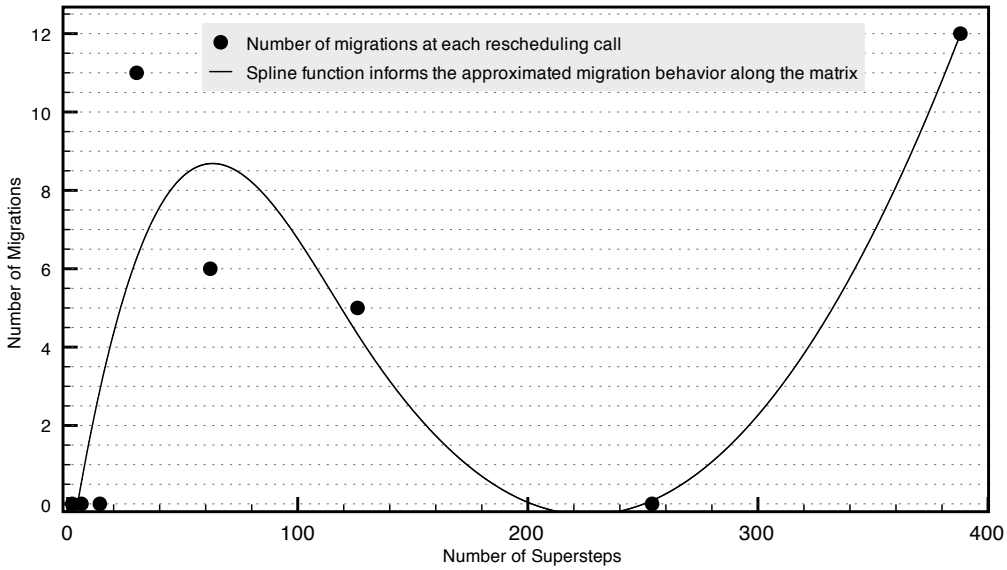


Figure 5.12: Migration behavior when testing a $200 \times 200$ matrix with initial $\alpha$ equal to 2

factory results were obtained with processes migration. The system stays stable during all application execution. Despite having more than one process mapped to one processor, sometimes just a portion of them is responsible for computation at a specific moment. This occurs because the processes are mapped to matrix columns, while supersteps comprise the anti-diagonals of the matrix. Figure 5.12 illustrates the migrations behavior along the execution with $\alpha$ 2.

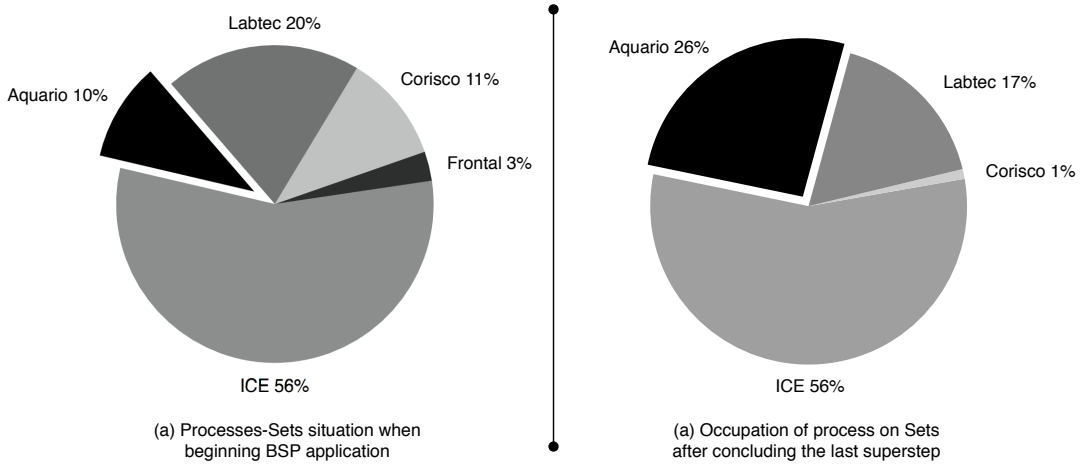


Figure 5.13: Different situations of processes location (in percentage) when testing a $200 \times 200$ matrix

We performed 8 calls for processes rescheduling when using $\alpha$ 2 and considering scenario iii on $200 \times 200$ matrix evaluation. No migrations were done at supersteps 2,6 and 14. The migration costs overlap the possible gains with processes relocation on these situations. Processes 21 up to 31 are migrated to cluster Aquario at superstep 30. Moreover, all processes from cluster Frontal are migrated to Aquario at superstep 62. Seven processes are candidates for migration at superstep 126: p30 to p36. However, only p31 up to p36 are migrated to cluster Aquario. These migrations happen because the processes

initially mapped to cluster Aquario do not collaborate yet with BSP computation. Migrations are not viable at superstep 254. In this moment, the initial processes mapped to cluster Aquario are active, increasing the cluster load as a whole and avoiding migrations to it. Finally, 12 processes (p189 to p200) are migrated to cluster Aquario when superstep 388 is crossed. At this time, all previous processes allocated to Aquario are inactive and the migrations are viable again. However, just 10 remaining supersteps are executed to amortize the processes migration costs. Figure 5.13 depicts both processes-resources mapping when starting and ending the treatment of the $200 \times 200$ matrix. We observed that slower clusters tend to loss processes to faster ones, since analyzed application may be classified as CPU-bound.

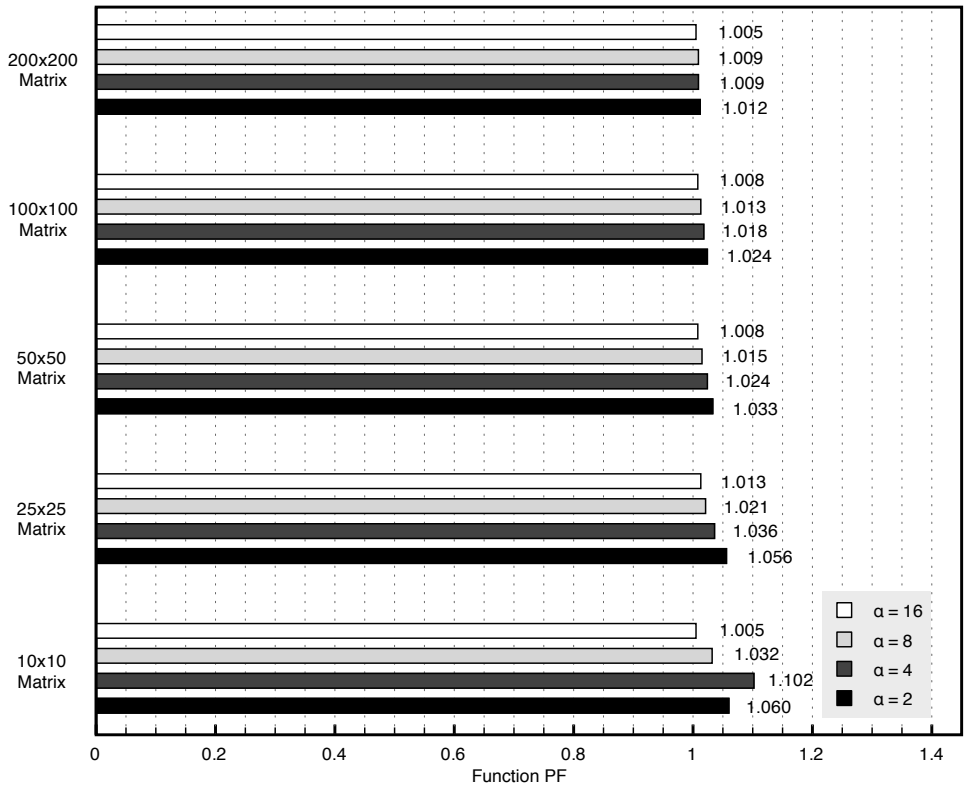


Figure 5.14: MigBSP overhead for Smith-Waterman BSP application. Performance Function (PF) on $x$ axis means $\frac{time\ in\ scenario\ ii}{time\ in\ scenario\ i}$

Lastly, Figures 5.14 and 5.15 describe the comparisons between scenarios i and ii and between the former with scenario iii, respectively. The larger is the matrix order, the lower is the MigBSP's overhead. This is explained because $\alpha$ increases at each rescheduling call. Thus, a larger number of superstep means a better opportunity to amortize the scheduling costs. While initial value of $\alpha$ does not present significant impact on the first comparison, its choice is important when testing MigBSP on scenario iii. Figure 5.15 shows $\alpha$ 4 as the best choice in average. An application with lower values of $\alpha$ tend to present more calls with migrations. On the other hand, first migrations are postponed with higher values of initial $\alpha$. Thus, their results may not outperform the migration costs.

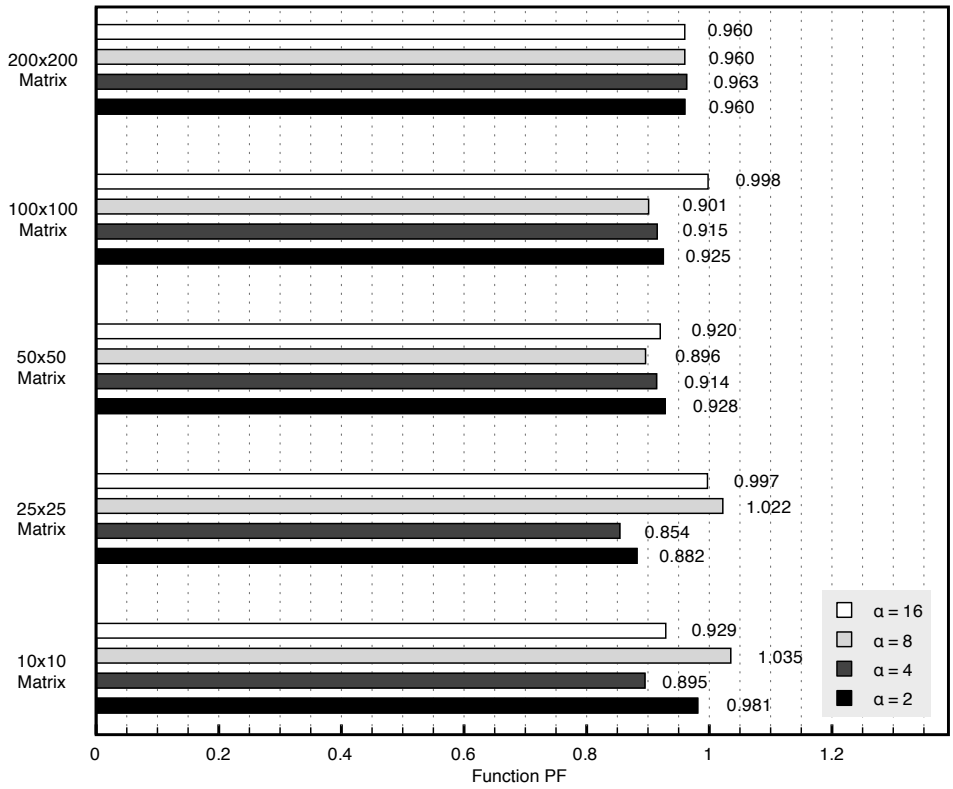


Figure 5.15: Gain or loss of performance when comparing both scenarios i and iii. Performance Function (PF) on $x$ axis means $\frac{time\ in\ scenario\ iii}{time\ in\ scenario\ i}$

## 5.6 LU Decomposition Application

Consider a system of linear equations $A.x = b$, where $A$ is a given $n \times n$ non singular matrix, $b$ a given vector of length $n$, and $x$ the unknown solution vector of length $n$. One method for solving this system is by using LU Decomposition technique. This technique comprises the decomposition of the matrix $A$ into a lower triangular matrix $L$ and an upper triangular matrix $U$ such that $A = LU$. A $n \times n$ matrix $L$ is called unit lower triangular if $l_{i,i} = 1$ for all $i, 0 \leq i < n$, and $l_{i,j} = 0$ for all $i, j$ where $0 \leq i < j < n$. An $n \times n$ matrix $U$ is called upper triangular if $u_{i,j} = 0$ for all $i, j$ with $0 \leq j < i < n$.

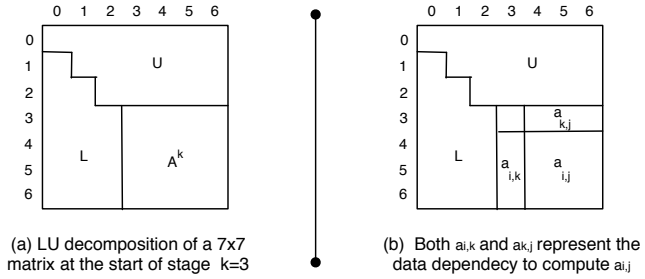


Figure 5.16: L and U matrixes decomposition using the same memory space of the original matrix $A^0$

On input, $A$ contains the original matrix $A^0$, whereas on output it contains the values of $L$ below the diagonal and the values of $U$ above and on the diagonal such that $LU = A^0$.

Figure 5.16 (a) illustrates the organization of LU computation. The values of $L$ and $U$ computed so far and the computed sub-matrix $A^k$ may be stored in the same memory space of $A^0$. Algorithm 8 presents a sequential algorithm for producing $L$ and $U$ in stages. Stage $k$ first computes the elements $u_{k,j}$, $j \geq k$, of row $k$ of $U$ and the elements $l_{i,k}$, $i > k$, of column $k$ of $L$. Then, it computes $A^{k+1}$ in preparation for the next stage. Algorithm 9 presents the functioning of the previous algorithm using just the elements from matrix $A$. Figure 5.16 (b) presents the data that is necessary to compute $a_{i,j}$ in the last statement of the Algorithm 9. Besides its own value, $a_{i,j}$ is updated using a value from the same line and another from the same column.

### 5.6.1 Modeling

This section explains how we modeled the LU sequential application on a BSP-based parallel one. Firstly, the bulk of the computational work in stage $k$ of the sequential algorithm is the modification of the matrix elements $a_{i,j}$ with $i, j \geq k+1$. Aiming to prevent communication of large amounts of data, the update of $a_{i,j} = a_{i,j} + a_{i,k}.a_{k,j}$ must be performed by the process whose contains $a_{i,j}$. This implies that only elements of column $k$ and row $k$ of $A$ need to be communicated in stage $k$ in order to compute the new sub-matrix $A^(k)$.

An important observation is that the modification of the elements in row $A(i,k+1 : n-1)$ uses only one value of column $k$ of $A$, namely $a_{i,k}$. The provided notation $A(i,k+1 :$

**Algorithm 8** Algorithm for LU Decomposition

```
1:  for k=0 to n-1 do
2:     for j=k to n-1 do
3:        u_{k,j} = a^k_{k,j}
4:     end for
5:     for i=k+1 to n-1 do
6:        l^k_{i,k} = a^k_{i,k} / u_{k,k}
7:     end for
8:     for i=k+1 to n-1 do
9:        for j-k+1 to n-1 do
10:          a^{k+1}_{i,j} = a^k_{i,j} - l_{i,k} . u_{k,j}
11:       end for
12:    end for
13: end for
```

**Algorithm 9** Algorithm for LU Decomposition using the same matrix A

```
1:  for k=0 to n-1 do
2:     for i=k+1 to n-1 do
3:        a_{i,k} = a_{i,k} / a_{k,k}
4:     end for
5:     for i=k+1 to n-1 do
6:        for j-k+1 to n-1 do
7:           a_{i,j} = a_{i,j} - a_{i,k} . a_{k,j}
8:        end for
9:     end for
10: end for
```

$n-1$) denotes the cells of line $i$ varying from column $k+1$ to $n-1$. If we distribute each matrix row over a limit set of $N$ processes, then the communication of an element from column $k$ can be restricted to a multicast to $N$ processes. Similarly, the modification of the elements in column $A(k+1,n:-1,j)$ uses only one value from row $k$ of $A$, namely $a_{k,j}$. If we distributed each matrix column over a limit set of $M$ processes, then the communication of an element of row $k$ can be restricted to a multicast to $M$ processes.

Considering the statements above, we are using a Cartesian scheme for the distribution of matrices. The square cyclic distribution is used as particularly suitable for matrix computations such as LU decomposition(BISSELING, 2004). For them, it is natural to organize the processes by two-dimensional identifiers $P(s,t)$ with $0 \leq s < M$ and $0 \leq t < N$, where the number of processes $p = M.N$. Figure 5.17 depicts a $6 \times 6$ matrix mapped to 6 processes, where $M = 2$ and $N = 3$. Assuming that $M$ and $N$ are multiple $n$, each process will store $nc$ (number of cells) cells in memory (see Equation 5.1).

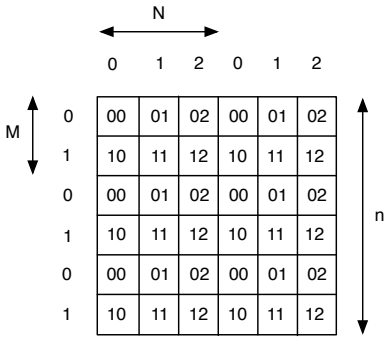


Figure 5.17: Cartesian distribution of a $6\times6$ ($n \times n$) matrix over $2\times3$ ($M \times N$) processors. The label "*st*" in the cell denotes its owner, process $P(s,t)$

$$nc = \frac{n}{M}.\frac{n}{N} \tag{5.1}$$

A parallel algorithm uses data parallelism for computations and the need-to-know principle to design the communication phase of each superstep. Following the concepts of BSP, all communication performed during a superstep will be completed when finishing it and the data will be available at the beginning of the next superstep(BONORDEN, 2007). Concerning this, we modeled our algorithm using three kinds of supersteps. They are explained in Table 5.9. The element $a_{k,k}$ is passed to the process that computes $a_{i,k}$ in the first kind of superstep.

The computation of $a_{i,k}$ is expressed in the beginning of the second superstep. This superstep is also responsible for sending the elements $a_{i,k}$ and $a_{k,j}$ to $a_{i,j}$. First of all, we pass the element $a_{i,k}$, $k+1 \leq i < n$, to the $N-1$ processes that execute on the respective row $i$. This kind of superstep also comprises the passing of $a_{k,j}$, $k+1 \leq j < n$, to $M-1$ processes which execute on the respective column $j$. The superstep 3 considers the computation of $a_{i,j}$, the increase of $k$ (next stage of the algorithm) and the transmission of $a_{k,k}$ to $a_{i,k}$ elements ($k+1 \leq i < n$). The BSP application will execute one superstep of type 1 and will follow with the interleaving of supersteps 2 and 3. Concerning this, a $n \times n$ matrix will trigger $2n+1$ supersteps in our LU modeling.

We modeled the Cartesian distribution $M \times N$ in the following manner: (*i*) $5 \times 5$ for 25 processes; (*ii*) $10 \times 5$ for 50 processes; (*iii*) $10 \times 10$ for 100 processes and; (*iv*) $20 \times 10$ for 200 processes. In addition, we are using square matrices with order 500, 1000, 2000 and 5000. Lastly, initial tests were executed using 4, 3 and 0.5 for $\alpha$, $\omega$ and $D$. Similarly

Table 5.9: Modeling three types of supersteps

| Type of superstep | Steps and explanation |
|---|---|
| First | Step 1.1 : $k = 0$ |
| | Step 1.2 - Pass the element $a_{k,k}$ to cells which will compute $a_{i,k}$ $(k+1 \leq i < n)$ |
| Second | Step 2.1 : Computation of $a_{i,k}$ $(k+1 \leq i < n)$ by cells which own them |
| | Step 2.2 : For each $i$ $(k+1 \leq i < n)$, pass the element $a_{i,k}$ to other $a_{i,j}$ elements in the same line $(k+1 \leq j < n)$ |
| | Step 2.3 : For each $j$ $(k+1 \leq j < n)$, pass the element $a_{k,j}$ to other $a_{i,j}$ elements in the same column $(k+1 \leq i < n)$ |
| Third | Step 3.1 : For each $i$ and $j$ $(k+1 \leq i, j < n)$, calculate $a_{i,j}$ as $a_{i,j} + a_{i,k}.a_{k,j}$ |
| | Step 3.2 : $k = k+1$ |
| | Step 3.3 : Pass the element $a_{k,k}$ to cells which will compute $a_{i,k}$ $(k+1 \leq i < n)$ |

to Smith-Watermann application, the tests with BSP-based LU application used heuristic one to choose the candidates for migration with $x$ equal to 80%.

### 5.6.2 Results

Table 5.10 presents the results when evaluating the $500 \times 500$, $1000 \times 1000$ and $2000 \times 2000$ matrices. The tests with the first matrix size show the worst results. Firstly, the higher the number of processes, the worse the performance, as we can observe in scenario i. The overhead related to communication and synchronization as well as the short computing grain are the reasons for the observed times. Secondly, MigBSP indicated that all migration attempts were inviable due to low computing and communication loads when compared to migration costs. Considering this, both scenarios ii and iii have the same time results. Even when there were candidates for migration with 25 processes, their transferring did not take place. $\alpha$ increased at each of the 8 rescheduling calls with this amount of processes. The execution when changing the number of processes reached negative values of $PM$, since Computation metric and $nc$ are reduced when increasing the number of processes (see Equation 5.1). Contrary to the execution with 25 processes, the system starts unbalanced when using 100 processes. Thus, $\alpha$ keeps the same value at the first three rescheduling calls (at supersteps 4, 8 and 12). MigBSP's adaptivity idea is useful in this situation, because $D$ is enlarged after $\omega$ calls without migration. This procedure causes an increase in the period of processes rescheduling, minimizing the model's interference on application execution. Thus, 11 calls for migration were done with 100 processes and a $500 \times 500$ matrix.

The first rescheduling call does not cause migrations when testing a $1000 \times 1000$ matrix with 25 processes. After this call at superstep 4, the next one at superstep 11 informs the migration of 5 processes from cluster Corisco. They were all transferred to cluster Aquario, which has the highest computation power. MigBSP does not point migrations in the future calls. $\alpha$ always doubles its value at each rescheduling call since the processes are balanced after the mentioned relocation. MigBSP obtained a gain of 12% of performance with this configuration when comparing scenarios i and iii. Maintaining this size of matrix and 50 processes, 6 processes from Frontal were migrated to Aquario at superstep

Table 5.10: Results when executing LU application linked to MigBSP middleware (time in seconds)

| Processes | 500×500 matrix | | | 1000×1000 matrix | | | 2000×2000 matrix | | |
|---|---|---|---|---|---|---|---|---|---|
| | Scen. i | Scen. ii | Scen. iii | Scen. i | Scen. ii | Scen. iii | Scen. i | Scen. ii | Scen. iii |
| 25 | 1.68 | 2.42 | 2.42 | 11.65 | 13.13 | 10.24 | 90.11 | 91.26 | 76.20 |
| 50 | 2.59 | 3.54 | 3.34 | 10.10 | 11.18 | 9.63 | 60.23 | 61.98 | 54.18 |
| 100 | 6.70 | 7.81 | 7.65 | 15.22 | 16.21 | 16.21 | 48.79 | 50.25 | 46.87 |
| 200 | 13.23 | 14.89 | 14.89 | 28.21 | 30.46 | 30.46 | 74.14 | 76.97 | 76.97 |

9. Although these migrations are profitable, they do not provide stability to the system and the processes remain unbalanced among the resources. Migrations are inviable in the next 3 calls at supersteps 15, 21 and 27. MigBSP launches our second adaptation on rescheduling frequency in order to alleviate its impact and $\alpha$ begins to grow until the end of the application. The tests with 50 processes obtained gains of just 5% with processes migration. This is explained by the fact that *nc* is decreased in this configuration when compared to the one with 25 processes. In addition, as the LU execution continues, the computation required at each superstep is reduced as well. Thus, the more advanced the execution, the lesser the gain with migrations. This happens since the communication cost from either cluster Labtec or Corisco with Aquario is higher than the cost observed with cluster Frontal. The tests with 100 and 200 processes do not present migrations because the forces that act in favor of migration (Computation and Communication metrics) are weaker than the Memory metric in all rescheduling calls.

The execution with a $2000 \times 2000$ matrix presents good results because the computing grain is increased. We observed a gain of 15% with processes relocation when testing 25 processes. All processes from cluster Corisco were migrated to Aquario in the first rescheduling call (at superstep 4). Thus, the application can take profit from this relocation in its beginning, when it demands larger computing cycles. The time for concluding LU application is reduced when passing from 25 to 50 processes as we can see in scenario i. However, the use of MigBSP resulted in lower gains. Scenario i presented 60.23s while scenario iii achieved 56.18s (9% of profit). When considering 50 processes, 6 processes were transferred from cluster Frontal to Aquario at superstep 4. The next call occurs at superstep 9, where 16 processes from cluster Corisco were elected as migration candidates to Aquario. Still, MigBSP indicated the migration of only 14 processes, since there were only 14 processors unoccupied in the target cluster. The execution of 100 processes presented the same migration behavior when 50 were tested. However, the performance gain was reduced to 4% with 100 processes given the reduction of computation load per process.

We observed that the higher the matrix order, the greater the gain with processes migration. Considering this, we evaluated a $5000 \times 5000$ matrix as can be seen in the Figure 5.18. MigBSP obtained performance improvements in all tests with scenario iii. The simple movement of all process from cluster Corisco to Aquario represented a gain of 19% when executing 25 processes. The tests with 50 processes obtained 852.31s and 723.64s for scenario i and iii, respectively. Scenario iii for this matrix size achieved the same migration behavior of the tests with the $2000 \times 2000$ matrix. However, the increase of matrix order represented a gain of 15% (order 5000) instead of 10% (order 2000). This analysis helps us to verify our previous hypothesis about performance gains when enlarging the

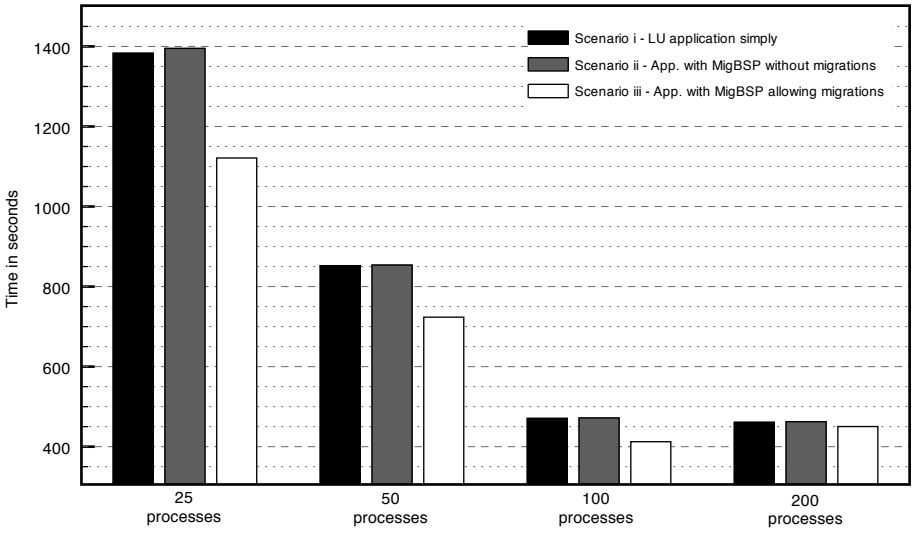


Figure 5.18: Performance graph for a $5000 \times 5000$ matrix

matrix. Finally, the tests with 200 processes indicated the migration of 6 processes (p195 up to p200) from cluster Corisco to Aquario at superstep 4. The nodes that belong to Corisco just execute one BSP process and nodes from Aquario begin to treat 2 processes with this transferring. The remaining rescheduling calls informs the processes that are executing on Labtec as those with the higher values of $PM$. However, their migration are not profitable. The final execution with 200 processes achieved 460.85s and 450.33s for scenarios i and iii, respectively.

## 5.7 Summary

This chapter presented our validation for MigBSP through using three BSP applications. Both applications and MigBSP scheduler were written with Simgrid simulator. This simulator was chosen since it turns the processes management easier and offers multi-cluster platforms creation. In addition, Simgrid is an active project, both in terms of research and development. Our evaluation encompasses the analysis of three different scenarios:(i) Application execution simply; (ii) Application execution with MigBSP without applying migrations and; (iii) Application linked to MigBSP scheduler when allowing migrations. We assembled an infrastructure with five Sets, where each node has a single processor. These Sets represent an adaptation of a real infrastructure with five clusters located at Federal University of Rio Grande do Sul, Brazil. Labtec, Corisco and Frontal clusters have their nodes linked by Fast Ethernet, while ICE and Aquario are clusters with a Gigabit connection. The migration costs were based on real executions with AMPI (HUANG et al., 2006).

Lattice-Boltzmann application evaluation showed us that when we have a short number of supersteps, we do not have time to outperform the employed costs in eventual migrations. This situation was observed when testing 10 and 50 supersteps. On the other hand, the bigger the number of supersteps the greater the gain with processes migration as we can verify with 2000 supersteps. After executing the migrations with 200 processes, the system remains unbalanced and after crossing 3 (value of tested $\omega$) we enlarge $D$. This procedure represents MigBSP's second adaptation, where we try to increase $\alpha$ even if the system is unstable in order to decrease MigBSP overhead on application execution.

Initially, MigBSP was developed to deal with processes that present a regular behavior. However, it allows to fill parameters that turn possible to adjust its functioning to treat irregularity in an efficient manner. In this context, we tested MigBSP over dynamic programming-based Smith-Waterman application, which the computing density increases when increasing the crossed supersteps as well. While each anti-diagonal of the matrix means a supersteps, we modeled each matrix row to be computed by a specific process. The main conclusions on this application were: (i) the option to migrate a percentage of processes instead just one was pertinent, since we can relocate all processes from a slower cluster to a faster one; (ii) a simple way to obtain performance is designing $\alpha$ in order to trigger the rescheduling call close to the end of the application, since MigBSP tends to select those processes that have more computational load to migrate; (iii) the greater is the load computational density of the last supersteps, the better will be the results with the migration of the last processes and; (iv) the application behavior implies that the processors can present variations on their load during the crossing of supersteps, changing their viability to receive processes.

Our last application comprises a BSP-version of the widely used LU decomposition. As observed on past tests, MigBSP showed again a low overhead when comparing scenarios *i* and *ii* of this application. For instance, it imposes 3% of cost with 50 processes and a $2000 \times 2000$ matrix. This feature is due to the simplicity of the *PM* (Potential of Migration) calculus as well as to the MigBSP adaptations regarding rescheduling frequency. *PM* evaluation considers the hierarchy organization with Sets and processes, not involving all processes-resources tests. MigBSP adaptations work to turn the model viable, especially when migrations cause performance gains but the system remains unbalanced. This occurred with a $1000 \times 1000$ matrix and 50 processes. The analysis of scenario *iii* showed us that the larger the matrix size, the bigger the gain with migrations in the first supersteps. Thus, the remaining supersteps may happen with a more optimized organization. The tests with a $5000 \times 5000$ matrix represent the best rate $\frac{scenario\ iii}{scenario\ i}$ in our results. Gains of 19% and 15% were obtained when executing 25 and 50 processes with migrations to the fastest cluster. Moreover, contrary to other situations, this matrix size enables migrations when using 200 processes due to its larger computing grain.

Considering the spectrum of these three applications, we can take the following conclusions in a nutshell: (i) the larger the computing grain, the better the gain with processes migration; (ii) MigBSP does not indicate the migration of those processes that have high migration costs when compared to computation and communication loads; (iii) MigBSP presented a low overhead on application execution when migrations are not applied; (v) our tests prioritizes migrations to cluster Aquario since it is the fastest one among considered clusters and tested applications are CPU-bound and; (vi) MigBSP does not work with previous knowledge about application. Thus, it indicates processes migration even when the application is close to finish. In this situation, migration brings an overhead since the remaining time for application conclusion is too short to amortize the considered costs.

# 6 CONCLUSION

BSP applications are composed by a set of processes that execute supersteps. Each superstep comprises both phases of computation and communication, ending with a barrier synchronization among the processes. This model does not specify how the processes must be mapped to the resources, leaving this issue to the user/developer. The time of each superstep is determined by the slowest process, due to the use of a synchronization barrier. Consequently, the processes-resources mapping is an important topic to achieve good performance when running BSP-based applications. Especially, this scheduling is yet more important on heterogeneous and dynamic environments like computational grids and multi-clusters.

Historically, the scheduling optimization was achieved by hand-tuning the software system to fit its needs on the computing environment. Although high optimization can be achieved, this process can be tedious and needs considerable expertise. Besides this, the hand-tuning mechanism was not portable across different computing environments and each different application required a new effort for processes scheduling. Despite the initial scheduling being strongly relevant to get better performance, the initial processes-processors mapping may not remain efficient during time. Thus, a possibility comprises the redistribution of processes during runtime through load rebalancing and processes migration. In this context, some initiatives use explicit migration calls that must be placed in application code(BHANDARKAR; BRUNNER; KALE, 2000; GALINDO; ALMEIDA; BADÍA-CONTELLES, 2008). In this technique the developer must still know details about both the parallel machine architecture and the application code. A different approach for load rebalancing happens at middleware level. With this, we can attempt to gain performance in an effortless way. Commonly, this approach does not require changes in the application code nor previous knowledge about the system. Therefore, the developer can focus his/her attention in the application correctness without worrying about migration decisions and implementation. Processes (re)allocation is hidden from the user, which will just perceive the changes in the execution time of the application.

This thesis presented a model for processes rescheduling called MigBSP. Its idea is to offer load (BSP processes) rebalancing among the resources in order to reduce the application's time. MigBSP acts at middleware level, without the need of any additional code in the application as well as without previous knowledge about both the application behavior and parallel machine organization. Our model employs self-adaptive strategies for processes rescheduling launching, examining the characteristics of the processes and choosing the software parameters needed to achieve high efficiency and low intrusion on application execution. Furthermore, MigBSP work with the idea of hierarchy with Sets and Set Managers abstractions. They are employed in order to reduce the spectrum of rescheduling tests when attempting for migration. Each Set can be understood as a site

like a cluster, local network or supercomputer. Each one presents a Set Managers which controls the data about BSP processes under its jurisdiction. The scheduling mechanism is located at each BSP process (inside communication and barrier functions) and at every Set Manager.

The adjustment provided by MigBSP occurs through the migration of those processes which have long computation time, perform several communication actions with other processes who belong to a same Set and present low migration costs. We opted to work with a global knowledge approach to choose the candidate processes for migration. Since the organization of BSP applications present barriers at each superstep natively, we use them to collect information from all process. With this, we don't have pay additional costs of synchronization. After chosen the candidates, each one is evaluate for migrating or not. This procedure involves the analysis of two times. The former considers that the tested process remains in the current resource. The second time includes the migration costs and represents an estimative of process' execution on the target processor.

## 6.1 Main Contributions

MigBSP offers a new approach for treating with processes migration on BSP applications. Under the scope of BSP model, MigBSP's contribution are threefold:

(i) Combination of three metrics - Computation, Communication and Memory - to evaluate the Potential of Migration (*PM*) of each BSP process at every rescheduling call;

(ii) Efficient adaptations that act on the periodicity of calls for processes rescheduling;

(iii) Employment of both Computation and Communication Patterns to analyze the regularity (stability) of each process on the instructions that it executes as well as on the communicated bytes at superstep.

The combination of the employed metrics demonstrated a pertinent strategy for decision making when choosing the candidate processes for migration. We analyzed metrics that act in favor of migration, representing the activity of the processes, and a metric that works in an opposite direction, meaning an idea of migration costs. While Computation and Communication metrics enter in the model in the former case, the Memory metric enters in the last case to observe migrations viability. These metrics are used to compute our decision function called Potential of Migration (*PM*). *PM* of a process $i$ to a target Set $j$ is reached according to Equation 6.1.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{6.1}$$

The higher the value of $PM(i,j)$, the higher the odds to migrate the involved process to the target Set. PM's concept is analogous to the idea of force from physics. A friction force (Memory metric) acts over a body (BSP processes) as a counterforce to any other force attracting the body. Any attracting force (Computation and Communication metrics) must exceed the friction in magnitude to actually move the body. In addition, other scientific idea behind *PM* calculus is the use of processes and Sets for testing migrations, not involving all possible processes-processors tests at the rescheduling moment. We employed this heuristic to obtain a satisfactory scheduling, both in terms of performance (makespan) and consumed time (efficiency). The use of a heuristic allows MigBSP to

achieve low overhead on application execution, while takes profitable decisions on BSP processes rescheduling BSP processes as revealed by Chapter 5.

MigBSP performs the call for migration testing after the barrier synchronization, between two supersteps. However, we agree that this call at each end of superstep is an onerous task. We based our assertion on the fact that migrations may not occur at each superstep and this configuration would lead to additional costs simply. Consequently, we developed two adaptations that manage an sliding interval between migrations attempts. This interval is controlled by a variable called $\alpha$. The adaptations' ideas are: (*i*) to postpone the rescheduling call if the system is stable (processes are balanced) or to turn it more frequent, otherwise; (*ii*) to delay this call if a pattern without migrations in $\omega$ calls is observed. Especially, this last adaptivity is suitable on situations in which profitable migrations take place but the processes remain unbalanced among the resources. In this situation, we enlarge $\alpha$ even if MigBSP detects unbalancing in the system. Thus, MigBSP can decrease its impact on application execution after performing migrations.

We establish that processes' regularity is important for us, because we expect that migrated processes perform a similar behavior on the destination place. Thus, we use Computation and Communication patterns in our equations to measure the regularity of the processes. The idea here is to observe the migrations behavior on dynamic applications. A process is regular on its computation phase if it performs an approximate number of instructions at each superstep. While the Computation pattern - $P_{comp}(i)$ - just considers a specific process $i$, Communication pattern - $P_{comm}(i,j)$- involves a process and a target Set. The regularity on communication phase treats the number of communicated bytes at each superstep. Both patterns are close to 1 if the process can be considered regular and close to 0 on the contrary. Before starting the BSP application, MigBSP assumes that all processes are stable. Still, the value of their patterns will decrease if this initial assumption is proved wrong.

## 6.2 Results Remarks

We studied and implemented the functioning of three applications in order to validate MigBSP. Both MigBSP and applications were tested through simulation with the Simgrid Simulator. We evaluated the following BSP applications: (i) Lattice-Boltzmann method that models fluid dynamics; (ii) Dynamic programming-based Smith-Waterman method used in DNA sequencing operations and; (iii) LU decomposition method which is employed to make equations solving easier. Firstly, we concluded that MigBSP presented a low overhead on application execution. The higher the number of supersteps the lower the relative overhead imposed by MigBSP. Taking into consideration all spectrum of evaluations, the mean overhead for our model is lower than 8%. For instance, expenses like 2.8%, 2.4% and 6.5% were found when testing 100 processes on the first, second and third BSP applications, respectively. These percentages were brought comparing the scenarios where the application was executed simply and that where it was executed with MigBSP without allowing migrations. These rates were reached due to both model's adaptations on rescheduling frequency and fast calculation of *PM*.

Other relevant point treats the executions where MigBSP does not point out migrations. MigBSP does not indicate the migration of those processes that have high migration costs when compared to computation and communication loads. For instance, the execution of Lattice-Boltzmann application with 10 processes does not produce any processes relocation. In this case, the processes run together in only one cluster and their

transferring to the fastest cluster are inviable. Migrations did not take place when using 200 processes and matrices order of 500, 1000 and 2000 on LU application too. In this case, the main reason is the short computing grain when managing matrices sizes like informed. Considering this, MigBSP adds an overhead on normal application execution inevitably.

Lastly, we must mention the situations where MigBSP represents a good option to get better performance. We tested MigBSP over an heterogeneous multi-cluster architecture. Thus, we observed that our model took decisions to migrate processes to the fastest cluster. The earlier the processes reassignment the greater the migration gains since the processes can execute in a more optimized fashion. For example, 14.7% and 11.6% of gains in time were verified when running the first application with 25 processes. The former percentage is related to execution with $\alpha$ equal to 4, which outperforms $\alpha$ 16 (the other rate) because the migrations were done earlier with lower value of $\alpha$ in this situation. Moreover, we analyzed that the larger the computing grain the larger the gain with processes migrations to faster resources on CPU-bound applications. The execution of LU application with MigBSP over a $5000 \times 5000$ matrix produces a reduction on application time up to 19%. The gain is limited to 15% when changing the matrix order to 2000. Furthermore, contrary to other situations in this application, the magnitude of 5000 enables migrations with performance profits when using 200 processes due to the larger computing load of the processes.

## 6.3 Future Works

The continuation of this thesis consists of works in the following directions:

(i) Use of a new heuristic to choose the processes that will be migrated;

(ii) Analysis of Computation and Communication patterns that act over a collection of supersteps instead of every superstep;

(iii) Evaluation of MigBSP with different observations of migration costs;

(iv) Test of MigBSP when changing the availability of the resources along the application execution.

$PM(i, j)$ of a candidate process $i$ is associated to a Set $j$. Currently, the manager of this Set will select the most suitable processor under its control to receive the process $i$. Other possible idea could be apply backtracking algorithm in order to evaluate which migrations are really viable. The algorithm assembles a tree with $q+1$ levels, where $q$ is the quantity of entries in the list of *PMs*. Formerly, we create an initial node (root) that represents the current scheduling. After that, we measure the performance of this node using the fitness function. This function will show the estimated time to execute the application given the parameters captured in runtime. From the root node we create $r$ other nodes, where $r$ is the number of processors in Set $j$. This Set is indicated by the first *PM* in the list of *PMs*. For each new node, we also calculate the fitness function and compared it with the rate of the father node. If the result of the child node is higher than the result of its father, this child node is eliminated from the tree automatically. The key idea is to abandon each partial candidate as soon as we can determine that it cannot be seen as a efficient solution.

For every following level $l$ ($l \geq 2$), we create $r$ new nodes for each father node, where $r$ is the number of processors in the Set $j$ indicated by $(l-1)^{th}$ $PM(i,j)$ in the list. Similarly, each new node is evaluated using the fitness function and it is eliminated according to rule described above. The Construction of the tree finishes when either there are not more *PMs* in the list or when all nodes in the last level were eliminated. Each node of the tree represents a new scheduling and comprises the modification of just one cell in scheduling matrix of its father. Concerning this, we have a space of pertinent schedulings that can occur in the system. The processes that will migrated are found in the following manner. The node with the lowest evaluation is found and a path is created from it up to the root node. Each node in this path corresponds a processor that will receive the process $i$ marked in the $PM(i,j)$ of the specific node level.

The second activity includes the observation of a regular behavior in application execution taking into account more than one superstep. For example, a process may execute 100, 150 and 200 instructions at the former three supersteps of the application. The following three superstep can repeat these magnitudes characterizing a regular behavior. The focus here is to find out a possible interval of supersteps where a regular behavior happens. Henceforth, MigBSP can increase the possibilities to migrate the considered process since it has higher odds to follow its current behavior on the target resource. The analysis of this branch of work will begin with the studies from Tanbeer, Ahmed and Jeong (TANBEER; AHMED; JEONG, 2009).

Our third direction for future work consists in evaluating MigBSP with different observations of migration costs. In other words, the idea here is to employ a different way for getting $MEM(i,j)$ metric. Our tests completed this metric based on executions with AMPI communication library. Therefore, we agree that the presented results showed MigBSP's behavior for this specific situation. Considering this, our idea is to fill out Memory metric field with migrations costs data from other migration libraries/mechanisms. In this way, we will analyze the previous work from Neves et al. (NEVES et al., 2007) which comprises an evaluation of processes migration using virtual machines. Finally, we know that large-scale grids present resources availability problems. Thus, we intend to evaluate MigBSP when modifying this issue. Simgrid offers availability treatment through files that describe the resources' state along the time.

# REFERENCES

AGGARWAL, G.; MOTWANI, R.; ZHU, A. The load rebalancing problem. In: SPAA '03: PROCEEDINGS OF THE FIFTEENTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 2003, New York, NY, USA. **Anais...** ACM Press, 2003. p.258–265.

AUMAGE, O.; HOFMAN, R.; BAL, H. NetIbis: an efficient and dynamic communication system for heterogeneous grids. In: CLUSTER COMPUTING AND GRID CONFERENCE, 2005, Cardiff, UK. **Proceedings...** IEEE Computer Society, 2005.

BATISTA, D. M.; FONSECA, N. L. S. da; MIYAZAWA, F. K.; GRANELLI, F. Self-adjustment of resource allocation for grid applications. **Comput. Netw.**, New York, NY, USA, v.52, n.9, p.1762–1781, 2008.

BEAUMONT, O.; ROBERT, A. L. Y. Static Scheduling Strategies for Heterogeneous Systems. **Computing and Informatics**, [S.l.], v.21, p.413–430, 2002.

BEGNUM, K. Xen Virtualization and Multi-host Management Using MLN. In: AIMS '07: PROCEEDINGS OF THE 1ST INTERNATIONAL CONFERENCE ON AUTONOMOUS INFRASTRUCTURE, MANAGEMENT AND SECURITY, 2007, Berlin, Heidelberg. **Anais...** Springer-Verlag, 2007. p.229–229.

BERMAN, F.; CASANOVA, H.; CHIEN, A.; COOPER, K.; DAIL, H.; DASGUPTA, A.; DENG, W.; DONGARRA, J.; JOHNSSON, L.; KENNEDY, K.; KOELBEL, C.; LIU, B.; LIU, X.; MANDAL, A.; MARIN, G.; MAZINA, M.; MELLOR-CRUMMEY, J.; MENDES, C.; OLUGBILE, A.; PATEL, J. M.; REED, D.; SHI, Z.; SIEVERT, O.; XIA, H.; YARKHAN, A. New grid scheduling and rescheduling methods in the GrADS project. **Int. J. Parallel Program.**, Norwell, MA, USA, v.33, n.2, p.209–229, 2005.

BHANDARKAR, M. A.; BRUNNER, R.; KALE, L. V. Run-Time Support for Adaptive Load Balancing. In: IPDPS '00: PROCEEDINGS OF THE 15 IPDPS 2000 WORKSHOPS ON PARALLEL AND DISTRIBUTED PROCESSING, 2000, London, UK. **Anais...** Springer-Verlag, 2000. p.1152–1159.

BISSELING, R. H. **Parallel Scientific Computation**: a structured approach using bsp and mpi. [S.l.]: Oxford University Press, 2004.

BOERES, C.; NASCIMENTO, A. P.; REBELLO, V. E. F.; SENA, A. C. 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, 2005, New York, NY, USA. **Anais...** ACM Press, 2005. p.1–6.

BONORDEN, O. Load Balancing in the Bulk-Synchronous-Parallel Setting using Process Migrations. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2007), 21., 2007. **Anais...** IEEE, 2007. p.1–9.

BONORDEN, O.; GEHWEILER, J.; HEIDE, F. M. auf der. Load Balancing Strategies in a Web Computing Environment. In: PROCEEEDINGS OF INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS (PPAM), 2005, Poznan, Poland. **Anais...** [S.l.: s.n.], 2005. p.839–846.

BOUKERCHE, A.; DAS, S. K. Dynamic load balancing strategies for conservative parallel simulations. In: PADS '97: PROCEEDINGS OF THE ELEVENTH WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 1997, Washington, DC, USA. **Anais...** IEEE Computer Society, 1997. p.20–28.

CAMARGO, R. Y. D.; KON, F.; GOLDMAN, A. Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In: SBAC-PAD '05: PROCEEDINGS OF THE 17TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE ON HIGH PERFORMANCE COMPUTING, 2005, Washington, DC, USA. **Anais...** IEEE Computer Society, 2005. p.226–234.

CAMERON, D.; CARVAJAL-SCHIAFFINO, R.; MILLAR, A.; NICHOLSON, C.; STOCKINGER, K.; ZINI, F. Analysis of Scheduling and Replica Optimisation Strategies for Data Grids using OptorSim. **Journal of Grid Computing**, [S.l.], v.2, n.1, p.57–69, March 2004.

CAO, J. Self-Organizing Agents for Grid Load Balancing. In: GRID '04: PROCEEDINGS OF THE FIFTH IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING (GRID'04), 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.388–395.

CASANOVA, H. Distributed computing research issues in grid computing. **SIGACT News**, New York, NY, USA, v.33, n.3, p.50–70, 2002.

CASANOVA, H.; LEGRAND, A.; QUINSON, M. SimGrid: a generic framework for large-scale distributed experiments. In: TENTH INTERNATIONAL CONFERENCE ON COMPUTER MODELING AND SIMULATION (UKSIM), 2008, Los Alamitos, CA, USA. **Anais...** IEEE Computer Society, 2008. p.126–131.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.14, n.2, p.141–154, 1988.

CHA, H.; LEE, D. H-BSP: a hierarchical bsp computation model. **J. Supercomput.**, Hingham, MA, USA, v.18, n.2, p.179–200, 2001.

CHAUBE, R.; CARINO, R. L.; BANICESCU, I. Effectiveness of a Dynamic Load Balancing Library for Scientific Applications. In: ISPDC '07: PROCEEDINGS OF THE SIXTH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING, 2007, Washington, DC, USA. **Anais...** IEEE Computer Society, 2007. p.32.

CHEN, C.; TONG, W. The Application of the BSP Model on DataGrid. In: SCC '04: PROCEEDINGS OF THE 2004 IEEE INTERNATIONAL CONFERENCE ON SERVICES COMPUTING, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.471–474.

CHEN, L.; WANG, C.-L.; LAU, F. Process reassignment with reduced migration cost in grid load rebalancing. **Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on**, [S.l.], p.1–13, April 2008.

CHEN, P.-C.; LIN, C.-I.; HUANG, S.-W.; CHANG, J.-B.; SHIEH, C.-K.; LIANG, T.-Y. A Performance Study of Virtual Machine Migration vs. Thread Migration for Grid Systems. In: AINAW '08: PROCEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS - WORKSHOPS, 2008, Washington, DC, USA. **Anais...** IEEE Computer Society, 2008. p.86–91.

CHEUNG, A. K. Y.; JACOBSEN, H.-A. Dynamic Load Balancing in Distributed Content-Based Publish/Subscribe. In: INTERNATIONAL MIDDLEWARE CONFERENCE, 7., 2006. **Anais...** Springer, 2006. p.141–161. (Lecture Notes in Computer Science, v.4290).

DOBBER, M.; MEI, R. van der; KOOLE, G. Dynamic Load Balancing and Job Replication in a Global-Scale Grid Environment: a comparison. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.20, n.2, p.207–218, 2009.

DONGARRA, J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An Introduction to the MPI Standard**. Knoxville, USA: University of Tennessee, Knoxville, 1995. (Technical report, CS-95-274).

DU, C.; GHOSH, S.; SHANKAR, S.; SUN, X.-H. A Runtime System for Autonomic Rescheduling of MPI Programs. In: ICPP '04: PROCEEDINGS OF THE 2004 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.4–11.

DU, C.; SUN, X.-H.; WU, M. Dynamic Scheduling with Process Migration. In: CCGRID '07: PROCEEDINGS OF THE SEVENTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2007, Washington, DC, USA. **Anais...** IEEE Computer Society, 2007. p.92–99.

DUTOT, P.; GOLDMAN, A.; KON, F.; NETTO, M. Scheduling moldable BSP tasks. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 11., 2005. **Anais...** Springer Verlag, 2005. p.157–172. (Lecture Notes in Computer Science, v.3834).

ELSASSER, R.; MONIEN, B.; PREIS, R. Diffusive load balancing schemes on heterogeneous networks. In: SPAA '00: PROCEEDINGS OF THE TWELFTH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 2000, New York, NY, USA. **Anais...** ACM Press, 2000. p.30–38.

FERNANDES, R.; PINGALI, K.; STODGHILL, P. Mobile MPI programs in computational grids. In: PPOPP '06: PROCEEDINGS OF THE ELEVENTH ACM SIGPLAN

SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Anais...** ACM, 2006. p.22–31.

FIEGE, L.; CILIA, M.; MUHL, G.; BUCHMANN, A. Publish-Subscribe Grows Up: support for management, visibility control, and heterogeneity. **IEEE Internet Computing**, Piscataway, NJ, USA, v.10, n.1, p.48–55, 2006.

FOSTER, I.; KESSELMAN, C. **The Grid**: blueprint for a new computing infrastructure. 2nd ed. San Francisco, CA, USA: Morgan Kaufmann, 2003. 800p.

FRACHTENBERG, E.; SCHWIEGELSHOHN, U. New Challenges of Parallel Job Scheduling. **Job Scheduling Strategies for Parallel Processing**, [S.l.], v.4942, p.1–23, May 2008.

GALINDO, I.; ALMEIDA, F.; BADÍA-CONTELLES, J. M. Dynamic Load Balancing on Dedicated Heterogeneous Systems. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 15TH EUROPEAN PVM/MPI USERS' GROUP MEETING, DUBLIN, IRELAND, SEPTEMBER 7-10, 2008. PROCEEDINGS, 2008. **Anais...** Springer, 2008. p.64–74. (Lecture Notes in Computer Science, v.5205).

GARCÍA, E. W.; MORALES-LUNA, G. Simulation for bulk synchronous parallel superstep task assignment in desktop grids characterised by gaussian parameter distributions. **Multiagent and Grid Systems**, [S.l.], v.4, n.2, p.141–166, 2008.

GEHWEILER, J.; SCHOMAKER, G. Distributed Load Balancing in Heterogeneous Peer-to-Peer Networks for Web Computing Libraries. In: DS-RT '06: PROCEEDINGS OF THE 10TH IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2006, Washington, DC, USA. **Anais...** IEEE Computer Society, 2006. p.51–62.

GODFREY, P. B.; KARP, R. M. On the price of heterogeneity in parallel systems. In: SPAA '06: PROCEEDINGS OF THE EIGHTEENTH ANNUAL ACM SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2006, New York, NY, USA. **Anais...** ACM Press, 2006. p.84–92.

GOLDCHLEGER, A.; GOLDMAN, A.; HAYASHIDA, U.; KON, F. The implementation of the BSP parallel computing model on the InteGrade Grid middleware. In: MGC '05: PROCEEDINGS OF THE 3RD INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING, 2005, New York, NY, USA. **Anais...** ACM Press, 2005. p.1–6.

GOLDCHLEGER, A.; KON, F.; GOLDMAN, A.; FINGER, M.; BEZERRA, G. C. InteGrade object-oriented Grid middleware leveraging the idle computing power of desktop machines: research articles. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, UK, v.16, n.5, p.449–459, 2004.

GROEN, D.; HARFST, S.; ZWART, S. P. On the Origin of Grid Species: the living application. In: COMPUTATIONAL SCIENCE - ICCS 2009, 9TH INTERNATIONAL CONFERENCE, BATON ROUGE, LA, USA, MAY 25-27, 2009, PROCEEDINGS, PART I, 2009. **Anais...** Springer, 2009. p.205–212. (Lecture Notes in Computer Science, v.5544).

HAO, X.; DAI, Y.; ZHANG, B.; CHEN, T. Task Migration Enabling Grid Workflow Application Rescheduling. In: ASIA-PACIFIC WEB CONFERENCE, APWEB 2008, 10., 2008. **Anais...** Springer, 2008. p.130–135. (Lecture Notes in Computer Science, v.4976).

HARCHOL-BALTER, M.; DOWNEY, A. B. Exploiting process lifetime distributions for dynamic load balancing. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.15, n.3, p.253–285, 1997.

HEISS, H.-U.; SCHMITZ, M. Decentralized dynamic load balancing: the particles approach. **Inf. Sci. Inf. Comput. Sci.**, New York, NY, USA, v.84, n.1-2, p.115–128, 1995.

HERNANDEZ, I.; COLE, M. Scheduling DAGs on Grids with Copying and Migration. In: PPAM, 2007. **Anais...** Springer, 2007. p.1019–1028. (Lecture Notes in Computer Science, v.4967).

HUANG, C.; ZHENG, G.; KALé, L.; KUMAR, S. Performance evaluation of adaptive MPI. In: PPOPP '06: PROCEEDINGS OF THE ELEVENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2006, New York, NY, USA. **Anais...** ACM Press, 2006. p.12–21.

JENKS, S.; GAUDIOT, J.-L. An Evaluation of Thread Migration for Exploiting Distributed Array Locality. In: HPCS '02: PROCEEDINGS OF THE 16TH ANNUAL INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS, 2002, Washington, DC, USA. **Anais...** IEEE Computer Society, 2002. p.190.

JIANG, Y.; TONG, W.; ZHAO, W. Resource Load Balancing Based on Multi-agent in ServiceBSP Model. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE (3), 2007. **Anais...** Springer, 2007. p.42–49. (Lecture Notes in Computer Science, v.4489).

JIN, S.; SCHIAVONE, G.; TURGUT, D. A performance study of multiprocessor task scheduling algorithms. **J. Supercomput.**, Hingham, MA, USA, v.43, n.1, p.77–97, 2008.

KANG, P.; SELVARASU, N. K.; RAMAKRISHNAN, N.; RIBBENS, C. J.; TAFTI, D. K.; VARADARAJAN, S. Modular, Fine-Grained Adaptation of Parallel Programs. In: ICCS '09: PROCEEDINGS OF THE 9TH INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE, 2009, Berlin, Heidelberg. **Anais...** Springer-Verlag, 2009. p.269–279.

KONDO, D.; CASANOVA, H.; WING, E.; BERMAN, F. Models and Scheduling Mechanisms for Global Computing Applications. In: IPDPS '02: PROCEEDINGS OF THE 16TH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 2002, Washington, DC, USA. **Anais...** IEEE Computer Society, 2002. p.79.2.

KOVACS, J.; KACSUK, P. A Migration Framework for Executing Parallel Programs in the Grid. In: EUROPEAN ACROSS GRIDS CONFERENCE, 2004. **Anais...** Springer, 2004. p.80–89. (Lecture Notes in Computer Science, v.3165).

KRAUTER, K.; BUYYA, R.; MAHESWARAN, M. A taxonomy and survey of grid resource management systems for distributed computing. **Software Practice & Experience**, New York, NY, USA, v.32, n.2, p.135–164, 2002.

KRIVOKAPIC, N.; ISLINGER, M.; KEMPER, A. Migrating Autonomous Objects in a WAN Environment. **Jorunal of Intelligent Information Systems**, Hingham, MA, USA, v.15, n.3, p.221–251, 2000.

KWOK, Y.-K.; CHEUNG, L.-S. A new fuzzy-decision based load balancing system for distributed object computing. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.64, n.2, p.238–253, 2004.

LEE, H. Parallel Hashing Algorithms on BSP and QSM Models. **Parallel and Distributed Processing Symposium, International**, Los Alamitos, CA, USA, v.8, p.175b, 2004.

LEGRAND, A.; RENARD, H.; ROBERT, Y.; VIVIEN, F. Load-balancing iterative computations on heterogeneous clusters with shared communication links. In: PPAM-2003: FIFTH INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS, 2003. **Anais...** Springer Verlag, 2003. p.930–937. (LNCS 3019).

LI, Y.; LAN, Z. A Survey of Load Balancing in Grid Computing. In: COMPUTATIONAL AND INFORMATION SCIENCE, FIRST INTERNATIONAL SYMPOSIUM, CIS 2004, 2004. **Anais...** Springer, 2004. p.280–285. (Lecture Notes in Computer Science, v.3314).

LI, Y.; LAN, Z. A novel workload migration scheme for heterogeneous distributed computing. In: CCGRID '05: PROCEEDINGS OF THE FIFTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'05) - VOLUME 2, 2005, Washington, DC, USA. **Anais...** IEEE Computer Society, 2005. p.1055–1062.

LIU, X.; CHIEN, A. A. Realistic Large-Scale Online Network Simulation. In: SC '04: PROCEEDINGS OF THE 2004 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.31.

LOW, M. Y. H. Dynamic Load-Balancing for BSP Time Warp. In: SS '02: PROCEEDINGS OF THE 35TH ANNUAL SIMULATION SYMPOSIUM, 2002, Washington, DC, USA. **Anais...** IEEE Computer Society, 2002. p.267.

LOW, M. Y.-H.; LIU, W.; SCHMIDT, B. A Parallel BSP Algorithm for Irregular Dynamic Programming. In: ADVANCED PARALLEL PROCESSING TECHNOLOGIES, 7TH INTERNATIONAL SYMPOSIUM, 2007. **Anais...** Springer, 2007. p.151–160. (Lecture Notes in Computer Science, v.4847).

LU, K.; SUBRATA, R.; ZOMAYA, A. Y. Towards Decentralized Load Balancing in a Computational Grid Environment. In: ADVANCES IN GRID AND PERVASIVE COMPUTING, FIRST INTERNATIONAL CONFERENCE, GPC 2006, TAICHUNG, TAIWAN, MAY 3-5, 2006, PROCEEDINGS, 2006. **Anais...** Springer, 2006. p.466–477. (Lecture Notes in Computer Science, v.3947).

LUMB, I.; SMITH, C. **Scheduling attributes and platform LSF**. Norwell, MA, USA: Kluwer Academic Publishers, 2004. p.171–182.

MARTIN, J. M. R.; TISKIN, A. V. Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid. In: COMMUNICATING PROCESS ARCHITECTURES 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p.219–226.

MARTÍNEZ-GALLAR, J.-P.; ALMEIDA, F.; GIMÉNEZ, D. Mapping in Heterogeneous Systems with Heuristic Methods. In: APPLIED PARALLEL COMPUTING. STATE OF THE ART IN SCIENTIFIC COMPUTING, 8TH INTERNATIONAL WORKSHOP, PARA 2006, UMEÅ, SWEDEN, JUNE 18-21, 2006, REVISED SELECTED PAPERS, 2007. **Anais...** Springer, 2007. p.1084–1093. (Lecture Notes in Computer Science, v.4699).

MCCOLL, W. F. Scalable Computing. In: COMPUTER SCIENCE TODAY: RECENT TRENDS AND DEVELOPMENTS, 1995. **Anais...** Springer-Verlag, 1995. v.1000, p.46–61.

MIAO, W.; TONG, W. Agent based ServiceBSP Model with Superstep Service for Grid Computing. **Sixth International Conference on Grid and Cooperative Computing (GCC 2007)**, Los Alamitos, CA, USA, v.00, p.255–260, 2007.

MILANéS, A.; RODRIGUEZ, N.; SCHULZE, B. State of the art in heterogeneous strong migration of computations. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, UK, v.20, n.13, p.1485–1508, 2008.

MILOJICIC, D. S.; DOUGLIS, F.; PAINDAVEINE, Y.; WHEELER, R.; ZHOU, S. Process Migration. **ACM Computing Surveys**, Cambridge, MA 02142, Sept. 2000.

MONTERO, R. S.; HUEDO, E.; LLORENTE, I. M. Grid Resource Selection for Opportunistic Job Migration. In: INTERNATIONAL EURO-PAR CONFERENCE, 9., 2003. **Anais...** Springer, 2003. p.366–373. (Lecture Notes in Computer Science, v.2790).

MORENO-VOZMEDIANO, R.; ALONSO-CONDE, A. B. Influence of Grid Economic Factors on Scheduling and Migration. In: HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE - VECPAR, 2005. **Anais...** Springer, 2005. p.274–287. (Lecture Notes in Computer Science, v.3402).

MUNCK, S. D.; VANMECHELEN, K.; BROECKHOVE, J. Improving the Scalability of SimGrid Using Dynamic Routing. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE - ICCS, 2009. **Anais...** Springer, 2009. p.406–415. (Lecture Notes in Computer Science, v.5544).

NAGARAJAN, A. B.; MUELLER, F.; ENGELMANN, C.; SCOTT, S. L. Proactive fault tolerance for HPC with Xen virtualization. In: ICS '07: PROCEEDINGS OF THE 21ST ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.23–32.

NETTO, M. A. S.; BUYYA, R. Rescheduling co-allocation requests based on flexible advance reservations and processor remapping. In: IEEE/ACM INTERNATIONAL CONFERENCE ON GRID COMPUTING, 9., 2008. **Proceedings...** IEEE, 2008. p.144–151.

NEVES, M. V.; ROSA RIGHI, R. da; MAILLARD, N.; NAVAUX, P. O. A. Impacto da Migração de Máquinas Virtuais de Xen na Execução de Programas MPI. In: OITAVO WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2007, Gramado - RS. **Anais...** [S.l.: s.n.], 2007. p.45–52.

NGUYEN, H. X.; FIGUEIREDO, D. R.; GROSSGLAUSER, M.; THIRAN, P. Balanced Relay Allocation on Heterogeneous Unstructured Overlays. In: CONFERENCE ON COMPUTER COMMUNICATIONS - INFOCOM, 2008. **Anais...** IEEE, 2008. p.126–130.

NICULESCU, V. Cost evaluation from specifications for BSP programs. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2006), 20., 2006. **Anais...** IEEE, 2006.

ORDUNA, J. M.; ARNAU, V.; RUIZ, A.; VALERO, R.; DUATO, J. On the Design of Communication-Aware Task Scheduling Strategies for Heterogeneous Systems. In: ICPP '00: PROCEEDINGS OF THE PROCEEDINGS OF THE 2000 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2000, Washington, DC, USA. **Anais...** IEEE Computer Society, 2000. p.391.

PILLA, L. L.; ROSA RIGHI, R. da; CARISSIMI, A.; NAVAUX, P.; HEISS, H.-U. Avaliação do Reescalonamento Adaptativo de Processos BSP. In: WORKSHOP EM DESEMPENHO DE SISTEMAS COMPUTACIONAIS E DE COMUNICAçãO - WPERFORMANCE, 2009. **Anais...** [S.l.: s.n.], 2009. v.VIII, p.2129 – 2144.

PILLA, L. L.; ROSA RIGHI, R. da; NAVAUX, P. O. A. Analisando a Migração de Processos MPI para seu Emprego em Aplicações BSP. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 8, 2008, Santa Cruz, RS. **Anais...** [S.l.: s.n.], 2008. p.181–184.

ROSA RIGHI, R. da; PILLA, L.; CARISSIMI, A.; NAVAUX, P. O. A. Controlling Processes Reassignment in BSP Applications. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBACPAD 2008), 20., 2008. **Anais...** IEEE Computer Society, 2008. p.37–44.

ROSA RIGHI, R. da; PILLA, L. L.; CARISSIMI, A.; NAVAUX, P.; HEISS, H.-U. MigBSP: a novel migration model for bulk-synchronous parallel processes rescheduling. **High Performance Computing and Communications, 10th IEEE International Conferenc e on**, Los Alamitos, CA, USA, v.0, p.585–590, 2009.

ROSA RIGHI, R. da; PILLA, L. L.; CARISSIMI, A. S.; NAVAUX, P. O.; HEISS, H.-U. Applying Processes Rescheduling over Irregular BSP Application. In: COMPUTATIONAL SCIENCE – ICCS 2009, 2009. **Anais...** Springer, 2009. p.213–223. (LNCS, v.5544).

SALEHI, M. A.; DELDARI, H. Grid load balancing using an echo system of intelligent ants. In: PDCN'06: PROCEEDINGS OF THE 24TH IASTED INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS, 2006, Anaheim, CA, USA. **Anais...** ACTA Press, 2006. p.47–52.

SALEHIE, M.; TAHVILDARI, L. Autonomic computing: emerging trends and open problems. In: DEAS '05: PROCEEDINGS OF THE 2005 WORKSHOP ON DESIGN AND EVOLUTION OF AUTONOMIC APPLICATION SOFTWARE, 2005, New York, NY, USA. **Anais...** ACM Press, 2005. p.1–7.

SCHEPKE CLAUDIO; MAILLARD, N. Performance Improvement of the Parallel Lattice Boltzmann Method Through Blocked Data Distributions. In: INTERNATIONAL

SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2007. SBAC-PAD 2007, 19., 2007. **Anais...** [S.l.: s.n.], 2007. p.71–78.

SCHNEIDER, J.; GEHR, J.; HEISS, H.-U.; FERRETO, T.; ROSE, C. D.; ROSA RIGHI, R. da; RODRIGUES, E. R.; MAILLARD, N.; NAVAUX, P. Design of a Grid workflow for a climate application. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS 2009 (ISCC 2009), 2009. **Anais...** IEEE, 2009. p.793–799.

SHAH, R.; VEERAVALLI, B.; MISRA, M. On the Design of Adaptive and Decentralized Load Balancing Algorithms with Load Estimation for Computational Grid Environments. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.18, n.12, p.1675–1686, 2007.

SKILLICORN, D. B.; HILL, J. M. D.; MCCOLL, W. F. Questions and Answers about BSP. **Scientific Programming**, [S.l.], v.6, n.3, p.249–274, Fall 1997.

SMITH, J. M. A survey of process migration mechanisms. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.22, n.3, p.28–40, 1988.

SONG, J.; TONG, W.; ZHI, X. ServiceBSP Model with QoS Considerations in Grids. In: ADVANCED WEB AND NETWORK TECHNOLOGIES, AND APPLICATIONS, 2006. **Anais...** Springer, 2006. p.827–834. (Lecture Notes in Computer Science, v.3842).

SPOONER, D. P.; JARVIS, S. A.; CAO, J.; SAINI, S.; NUDD, G. R. Local Grid Scheduling Techniques using Performance Prediction. **IEE Proc. Comp. Digit. Tech., Nice, France**, [S.l.], v.150, n.2, p.87–96, April 2003.

SULISTIO, A.; PODUVAL, G.; BUYYA, R.; THAM, C.-K. On incorporating differentiated levels of network service into GridSim. **Future Gener. Comput. Syst.**, Amsterdam, The Netherlands, The Netherlands, v.23, n.4, p.606–615, 2007.

SUNDERAM, V. PVM: a framework for parallel distributed computing. **Concurrency: Practice and Experience**, Chichester, UK, v.2, n.4, p.315–339, 1990.

TANBEER, S. K.; AHMED, C. F.; JEONG, B.-S. Parallel and Distributed Frequent Pattern Mining in Large Databases. **High Performance Computing and Communications, 10th IEEE International Conference on**, Los Alamitos, CA, USA, v.0, p.407–414, 2009.

TANENBAUM, A. **Computer Networks**. 4th.ed. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003. 912p.

THAIN, D.; TANNENBAUM, T.; LIVNY, M. Distributed computing in practice: the condor experience: research articles. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, UK, v.17, n.2-4, p.323–356, 2005.

TOPCUOUGLU, H.; HARIRI, S.; WU, M. you. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, NJ, USA, v.13, n.3, p.260–274, 2002.

TSE, S. S. Online Bicriteria Load Balancing Using Object Reallocation. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.20, n.3, p.379–388, 2009.

TSENG, L.-Y.; CHIN, Y.-H.; WANG, S.-C. A minimized makespan scheduler with multiple factors for Grid computing systems. **Expert Syst. Appl.**, Tarrytown, NY, USA, v.36, n.8, p.11118–11130, 2009.

UTRERA, G.; CORBALAN, J.; LABARTA, J. Dynamic Load Balancing in MPI Jobs. In: THE 6TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTING, 2005. **Anais...** [S.l.: s.n.], 2005.

VADHIYAR, S.; DONGARRA, J. Self Adaptability in Grid Computing. **Concurrency and Computation: Practice and Experience, Special Issue: Grid Performance**, [S.l.], v.17, n.2–4, p.235–257, 2005.

VADHIYAR, S. S.; DONGARRA, J. J. Self adaptivity in Grid computing: research articles. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, UK, v.17, n.2-4, p.235–257, 2005.

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, New York, NY, USA, v.33, n.8, p.103–111, 1990.

VASILEV, V. P. BSPGRID: variable resources parallel computation and multiprogrammed parallelism. **Parallel Processing Letters**, [S.l.], v.13, n.3, p.329–340, 2003.

WANG, C.; MUELLER, F.; ENGELMANN, C.; SCOTT, S. L. Proactive process-level live migration in HPC environments. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Piscataway, NJ, USA. **Anais...** IEEE Press, 2008. p.1–12.

WANG, X.; ZHU, Z.; DU, Z.; LI, S. Multi-cluster Load Balancing Based on Process Migration. In: ADVANCED PARALLEL PROCESSING TECHNOLOGIES, 7TH INTERNATIONAL SYMPOSIUM, APPT 2007, 2007. **Anais...** Springer, 2007. p.100–110. (Lecture Notes in Computer Science, v.4847).

WARAICH, S. S. Classification of Dynamic Load Balancing Strategies in a Network of Workstations. In: FIFTH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: NEW GENERATIONS, 2008. **Anais...** IEEE Computer Society, 2008. p.1263–1265.

WATTS, J.; TAYLOR, S. A Practical Approach to Dynamic Load Balancing. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.9, n.3, p.235–248, 1998.

WEIQIN, T.; JINGBO, D.; LIZHI, C. Design and Implementation of a Grid-Enabled BSP. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 3., 2003. **Anais...** IEEE Computer Society, 2003.

WILKINSON, B.; ALLEN, M. **Parallell Programming**: techniques and applications using networked workstations and parallel computers. Upper Sadle River, New Jersey: Prentice Hall, 1999. p.38–81.

WILLIAMS, T. L.; PARSONS, R. J. The Heterogeneous Bulk Synchronous Parallel Model. In: IPDPS '00: PROCEEDINGS OF THE 15 IPDPS 2000 WORKSHOPS ON PARALLEL AND DISTRIBUTED PROCESSING, 2000, London, UK. **Anais...** Springer-Verlag, 2000. p.102–108.

YAGOUBI, B.; MEDEBBER, M. A load balancing model for grid environment. **Computer and information sciences, 2007. iscis 2007. 22nd international symposium on**, [S.l.], p.1–7, Nov. 2007.

YAMIN, A. Escalonamento em Sistemas Paralelos e Distribuídos. In: PRIMEIRA ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 2001, Gramado, RS. **Anais...** [S.l.: s.n.], 2001. p.75–124.

YOUNG, L.; MCGOUGH, S.; NEWHOUSE, S.; DARLINGTON, J. Scheduling Architecture and Algorithms within the ICENI Grid Middleware. In: UK E-SCIENCE PROGRAM, 2003. **Anais...** [S.l.: s.n.], 2003. p.5–12.

YU, Z.; SHI, W. An Adaptive Rescheduling Strategy for Grid Workflow Applications. **ipdps**, Los Alamitos, CA, USA, v.0, p.115, 2007.

ZAKI, M. J.; LI, W.; PARTHASARATHY, S. Customized dynamic load balancing for a network of workstations. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.43, n.2, p.156–162, 1997.

ZHANG, B.-Y.; MO, Z.; YANG, G.; ZHENG, W. An Efficient Dynamic Load-Balancing Algorithm in a Large-Scale Cluster. In: DISTRIBUTED AND PARALLEL COMPUTING, 6TH INTERNATIONAL CONFERENCE ON ALGORITHMS AND ARCHITECTURES FOR PARALLEL PROCESSING, ICA3PP, MELBOURNE, AUSTRALIA, OCTOBER 2-3, 2005, PROCEEDINGS, 2005. **Anais...** Springer, 2005. p.174–183. (Lecture Notes in Computer Science, v.3719).

ZHANG, Y.; KOELBEL, C.; COOPER, K. Hybrid Re-scheduling Mechanisms for Workflow Applications on Multi-cluster Grid. **Cluster Computing and the Grid, IEEE International Symposium on**, Los Alamitos, CA, USA, v.0, p.116–123, 2009.

ZHENG, G.; HUANG, C.; KALé, L. V. Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.40, n.2, p.90–99, 2006.

ZHU, J.; TONG, W.; DONG, X. Agent Assisted ServiceBSP Model in Grids. In: GCC '06: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2006, Washington, DC, USA. **Anais...** IEEE Computer Society, 2006. p.17–21.