3

Process rescheduling: enabling performance by applying multiple metrics and efficient adaptations

Rodrigo da Rosa Righi
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada - Universidade do Vale do Rio dos Sinos
Brazil

Laércio Pilla, Alexandre Carissimi and Philippe Navaux
Programa de Pós-Graduação em Computação - Universidade Federal do Rio Grande do Sul
Brazil

Hans-Ulrich Heiss
Kommunikations- und Betriebssysteme - Technische Universität Berlin
Germany

1. Introduction

As grid technologies gain popularity, separate clusters and computers are increasingly being interconnected to create computing architectures for the processing of scientific and commercial applications (Sánchez et al., 2010). These constituent parts may be different from each other as well as be located either within a single organization or across various geographical sites. Concerning this, the task of allocating processes to processors on such architecture often becomes a problem requiring considerable effort. In order to fully exploit this kind of environments, the programmer must know both the machine architecture and the application code properly. Moreover, each new application requires another analysis for process scheduling.
Since both resource management and scheduling are key services for grid environments, issues like load balancing represent a common concern for most developers. Thus, a possibility is to explore the automatic load balancing at middleware level, linking the balancer tool with the programming library. For instance, an allocation scheme where the processes with longer computing times are mapped to faster machines can be used. On the other hand, this approach is not the best one for irregular applications and dynamic distributed environments, where a good processes-resources assignment performed in the beginning of the application may not remain efficient with time (Low et al., 2007). At this moment, it is not possible to recognize either the amount of computation of each process or the communication patterns among them. Besides fluctuations in the processes' computation and communication actions, the processors' load may vary and networks may become congested while the application is running. An alternative is to perform process rescheduling by applying the migration of the processes to new resources, offering runtime load balancing (Chen et al., 2008).

In this context, we developed a model called MigBSP. MigBSP controls process rescheduling on dynamic and heterogeneous environments, like multi-cluster ones. It acts over BSP (**Bulk Synchronous Parallel**) applications (Valiant, 1990) and works with the concept of hierarchy in two levels using Sets (considering different networks) and Set Managers. MigBSP's adjusts the conclusion of both local computation and global communication phases of BSP processes to be faster taking benefit from data collected at runtime. This adjustment happens through the migration of those processes whose have long computation time, perform several communication actions with other processes that belong to a same Set and present low migration costs. The use of the Computation, Communication and Memory (migration costs) metrics aims to offer performance and better quality for scheduling decisions. Besides multiple metrics, other keyword of MigBSP is adaptivity. Contrary to existing approaches (Bonorden et al., 2005; Vadhiyar & Dongarra, 2005), MigBSP performs the rescheduling launching according to the system state in order to reduce its impact on application execution.
The present chapter describes MigBSP and its novel ideas for process rescheduling. We developed three different BSP-based scientific applications in order to verify the impact and the efficiency of MigBSP's algorithms. Besides the choice of the applications, we modeled a multi-cluster infrastructure and varied the number of processes. Summarizing the results, we achieved a mean performance gain of 19% when applying process migration over our applications. Moreover, a mean overhead lower than 7% was observed when migrations are not applied (if the model decides that migrations are not recommended during all application execution). Therefore, the use of multiple metrics and efficient adaptations configure MigBSP as a viable solution when treating migration of BSP processes. Besides BSP, MigBSP's algorithms can be used in several other situations where load balancing takes place, such as in Web servers, data centers and synchronous computations in general (Bonorden et al., 2005).
The remaining of this chapter is organized as follows. Section 2 shows related work and some opportunities of research. Section 3 describes the rationales of MigBSP, emphasizing its contribution and novel ideas. Section 4 presents our evaluation methodology. Section 5 discusses the results and points out the performance gain/loss when executing MigBSP. Finally, the concluding remarks of the chapter are displayed in Section 6.

2. Related Work

GridWay resource broker treats with time and cost optimization on scheduling and migration areas (Moreno-Vozmediano & Alonso-Conde, 2005). Both migration mechanisms consider only data from CPU, like speed and load. (Bhandarkar, Brunner & Kale, 2000) presented a support for adaptive load balancing in MPI applications. Periodically, MPI application transfers control to the load balancer using a special call MPI_Migrate(). This mechanism implies in modifications in the application code. Besides this last work, a fixed period for rescheduling launching is also demonstrated in the following approaches (Hernandez & Cole, 2007; Utrera et al., 2005). Adaptive MPI (AMPI) (Huang et al., 2006) uses Charm++ framework to offer load balancing. Charm++ uses workload data and objects communication pattern to redistribute the workload at each load balancing time.
A system for autonomic rescheduling of MPI (Message Passing Interface) programs is presented in (Du et al., 2004). This work presents an extensible rule-based mechanism for policy making. When a policy is satisfied, its actions are done. Besides the consideration of monitored data, this system also uses an application description in order to estimate the execution time. Vadhiyar and Dongarra presented a migration framework and self-adaptivity in GrADS system (Vadhiyar & Dongarra, 2005). However, they computed the migration costs as a fixed

value. In addition, the gain with rescheduling is based on the remaining execution time prediction over a new specified resource. Thus, this framework must work with applications in which their parts and durations are known in advance.
(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. This work considers static information about the behavior of the tasks (number of instructions, interactions among the tasks and amount of memory). Furthermore, the migration of tasks is performed only to a neighbor node (direct connection in the processors graph). (Du, Sun & Wu, 2007) measured the migration costs at application runtime. For that, they described a model that considers the process, the memory, the I/O and the communication states. Nevertheless, these authors specify neither when to launch the process migration, nor which processes will be migrated actually. Kondo et al. (Kondo et al., 2002) described a client-server scheduling model for global computing. Their model measures the processor speed, the network bandwidth and the disk space to set the number of work units that can be sent to a client. However, these values are not combined and the minimum of them gives the number of work that the server will pass to a client.
Concerning the BSP scope, we can cite two works that present migration on BSP applications. The first one describes the PUBWCL library which aims to take profit of idle cycles from nodes around the Internet (Bonorden et al., 2005). PUBWCL can migrate a process during its computation phase as well as after the barrier. All proposed algorithms just use computation data about processes and the the nodes. Other work comprises an extension of PUB library to support migration (Bonorden, 2007). The author explains that a load balancer decides when to launch the process migration. Nevertheless, this issue is not addressed in (Bonorden, 2007). Bonorden proposed both a centralized and a distributed strategies for load balancing. In the first one, all nodes send data about their CPU power and load to a master node. The master verifies the least and the most loaded node and migrates one process between them. In the distributed approach, every node chooses c other nodes randomly and asks them for their load. One process is migrated if the minimum load of c analyzed nodes is smaller than the load of the node that is performing the test. The drawback of this strategy is that it can create a lot of messages among the nodes. Moreover, both strategies take into consideration neither the communication among the processes, nor the migration costs.

3. MigBSP: Process Rescheduling Model

A BSP application is divided in one or more supersteps, each one containing both computation and communication phases followed by a barrier synchronization. Since the barrier always wait for the slowest process, MigBSP's final objective is to adjust the processes' location in order to reduce the supersteps' times. Figure 1 (a) shows a superstep k of an application in which the processes are not balanced among the resources. Figure 1 (b) depicts the expected result with processes redistribution at the end of superstep k, which will influence the execution of the following supersteps. MigBSP offers automatic load balancing at middleware level, requiring no changes in the application code nor previous knowledge about the system/application. All necessary data for its functioning can be captured directly in both communication and barrier functions as well as in other sources like the operating system. The final result of MigBSP is a formalism that answers the following issues regarding process migration: (i) "When" to launch the mechanism for process migration; (ii) "Which" processes are candidates for migration and; (iii) "Where" to put the elected processes. We are not interested in the keyword "How", that treats the mechanism employed to perform migrations.

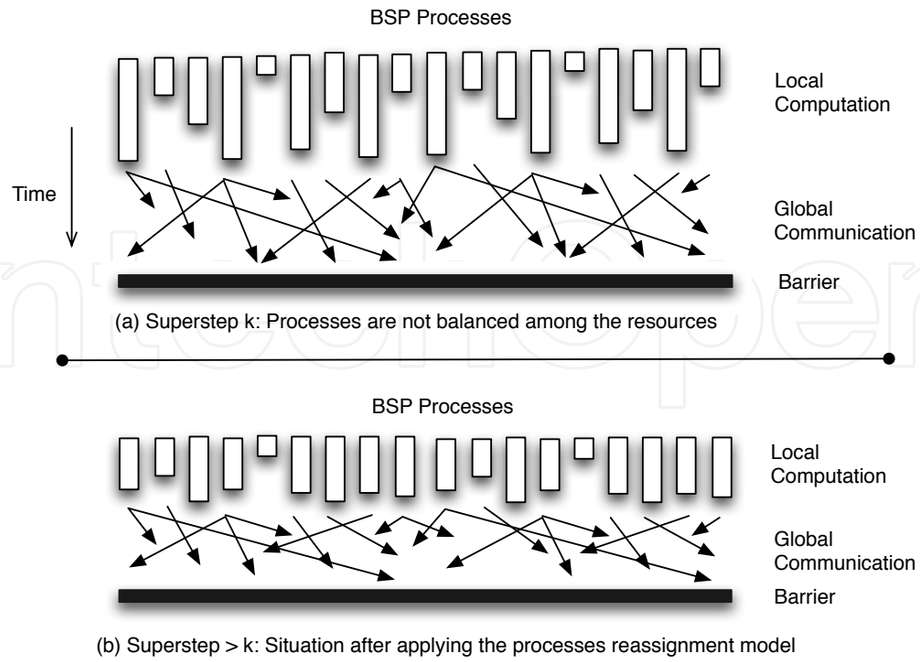


Fig. 1. BSP Supersteps in two different situations

MigBSP can be seen as a scheduling middleware. Concerning this area, (Casavant & Kuhl, 1988) proposed a scheduling taxonomy for general purpose distributed computing systems in order to formalize the classification of schedulers. MigBSP can be enclosed on the global and dynamic items. The dynamic feature considers that information for process scheduling are collected at application runtime. The role of scheduling is spread among several processes that cooperate among themselves in order to improve resource utilization (processors and 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.

3.1 Model of Parallel Machine and Communication

MigBSP works over an heterogeneous and dynamic distributed environment. The heterogeneous issue considers the processors' capacities (all processors have the same architecture, *e.g.* i386), as well as the network bandwidth and level (Fast and Gigabit Ethernet and multi-clusters environments, for instance). The dynamic behavior deals with environment changes which were perceived at runtime (such as network congestion and fluctuations on processors' load). Moreover, the dynamic behavior can also occur at process level, since some processes may need more computational power or increase their network interaction with other processes during application runtime. Each process is mapped to a real processor which can execute more than one process. In order to turn the scheduling more flexible and efficient, MigBSP proposes a hierarchical scheduling. The nodes are gathered to create the abstraction of a Set. A Set could be a LAN network or a cluster. Each Set is composed by one or more nodes (each with one or more processors) and a Set Manager. The scheduling mechanism is

located inside every process (additional code in barrier function) and inside each Set Manager. This last entity captures scheduling data from a Set and exchanges it among other managers. Our communication model affirms that the Sets are fully interconnected, meaning that there exists at least one communication path between any two nodes. The communication is asynchronous, where the sending is non blocking while the receiving is blocking. In addition, the underlying network protocol always guarantee reliability and the fact that the messages sent across the network are received in the order sent previously.

3.2 Question "When": Process Rescheduling Activation

The decision for process remapping is taken at the end of a superstep. We are employing the reactive migration approach (Milanés et al., 2008), where the process relocation is launched from outside the application (in this case, at middleware level). The migration point was chosen because in this moment it is possible to analyze data about the computation and communication from all processes. We applied two adaptations aiming to put as less intrusion in the application as possible. They provide an adaptable interval between migration calls.

3.2.1 First Adaptation: Controlling the Migration Interval based on the Processes' Balance

We are using an index α ($\alpha \in N^*$) in order to turn viable the adaptivity on process rescheduling calling. α will inform the interval between supersteps to apply process migration. This index increases if the system tends to the stability in the 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 α, 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 the times is computed. The distributed system is considered stable if both Inequalities 1 and 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. The D value is passed in model initialization. Figure 2 shows the algorithm that reveals how the α value is computed along the application execution. A variable called α' was employed to save the temporary value of α. α' will show the next interval to trigger the load balancing. α' suffers a variation of one unity at each superstep depending on the state of the system.

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

$$time\ of\ the\ fastest\ process > average\ time\ .\ (1 - D) \tag{2}$$

In Figure 2, t ($k \leq t \leq k + \alpha - 1$) is the index of a 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 α will have the same meaning in all algorithms). α' does not have an upper bound, but its lower value is the initial value of α. In the best case, the system is always in equilibrium and α' always increases. For example, if the system is always stable and the initial value of α is 10, after 10 supersteps the new value of α will be 20. The idea of the model is to minimize its intrusion in application execution while the system stays stable, postponing the process rescheduling activation according to α. In implementation view, BSP processes save their times in a vector and pass them to their Set Managers when rescheduling is activated. Following this, all Set Managers exchange their information. Taking into account the the times of each process, the Set Managers compute both Inequalities 1 and 2. Therefore, each manager knows the α' variation locally.

1. **for** t from superstep k to superstep $k + \alpha - 1$ **do**
2. **if** Inequalities 1 and 2 are true **then**
3. Increase α' by 1
4. **else if** $\alpha' >$ initial α
5. Decrease α' by 1
6. **end if**
7. **end for**
8. Call for BSP process rescheduling
9. $\alpha = \alpha'$

Fig. 2. Interval of supersteps α for the next call for BSP process rescheduling

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

The other adaptation considers the management of D (see Inequalities 1 and 2) based on the frequency of migrations. Figure 4 depicts the impact of D when defining the situation of the processes. The idea is to increase D if process rescheduling is activated for ω consecutive times and none migrations happen. The increase of D enlarges the interval in which the system is considered stable, causing the increase of α' consequently. In contrast, D can decrease down to a limit if each call produces the migration of at least one process. The algorithm depicted in Figure 3 presents how D is controlled at each rescheduling call.

1. $\gamma \leftarrow$ Consecutive rescheduling calls without migrations
2. **if** $\gamma \geq \omega$ **then**
3. **if** D + $\frac{D}{2} < 1$**then**
4. D $\leftarrow$ D + $\frac{D}{2}$
5. **end if**
6. **else if** D $>$ initial D and $\gamma = 0$ **then**
7. D $\leftarrow$ D - $\frac{D}{2}$
8. **end if**

Fig. 3. Stability of the system according to D

The computation of D is done by each Set Manager, which knows if migrations occurred during the migration call. 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 scheduling configuration. Consequently, it is pertinent to increase D in order to minimize MigBSP impact on application execution in this situation.

3.3 Question "Which": Choosing the Candidate Processes for Migration

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 specific Set. The key idea 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. The first two work at the computation and communication phases of a superstep. The Memory metric acts as an idea of migration costs.

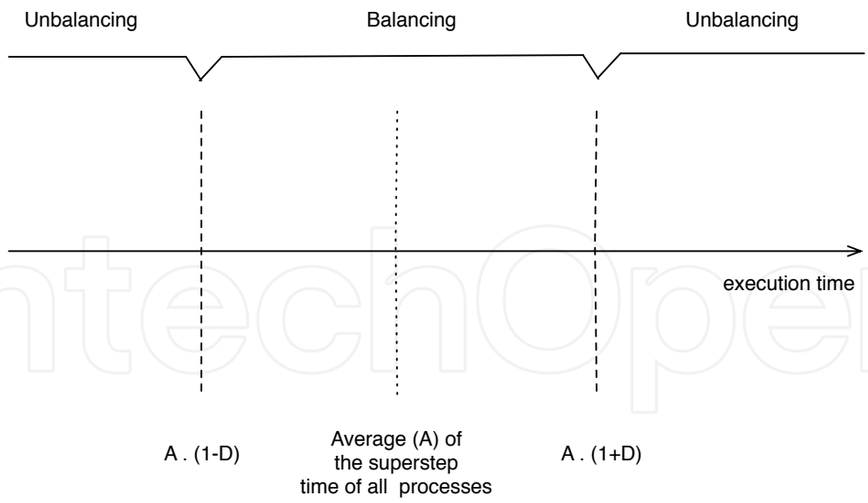


Fig. 4. Balancing situations which depend on the distance D from the average A

3.3.1 Computation Metric

Each process i computes $Comp(i,j)$ functions, where $Comp(i,j)$ informs the Computation metric for a process i and a specific Set j. Set j is used in $Comp(i,j)$ calculus to simulate the performance of process i on different sites of the parallel architecture. The data used to calculate this metric start at superstep k and finish at superstep $k+\alpha-1$. For every superstep t ($k \leq t \leq k+\alpha-1$), the number of processor's instructions (I_t) and the conclusion time 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 that belongs to the [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 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). 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 process 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 described in Figure 5. We consider the system stable if the forecast is within a δ margin of fluctuation from the amount of

instructions performed. For instance, if δ 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.

1. **for** t from superstep k to superstep $k+\alpha-1$ **do**
2. **if** $PI_t(i) \geq I_t(i).(1-\delta)$ and $PI_t(i) \leq I_t(i).(1+\delta)$ **then**
3. Increases $P_{comp}(i)$ by $\frac{1}{\alpha}$ up to 1
4. **else**
5. Decreases $P_{comp}(i)$ by $\frac{1}{\alpha}$ down to 0
6. **endif**
7. **endfor**

Fig. 5. Computation Pattern $P_{comp}(i)$ of process i

The computation pattern $P_{comp}(i)$ is an element in the $Comp(i,j)$ function. Other element is a computation time prediction $CTP_{k+\alpha-1}(i)$ of the process i at superstep $k+\alpha-1$ (last superstep executed before process rescheduling). Analogous to PI prediction, CTP also works with the 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 the $k+\alpha-1^{th}$ superstep. For each processor in a Set, its load is multiplied by its theoretical capacity. Concerning this, the Set Managers compute a performance average of their Sets and exchange this value. Each manager calculates $ISet(j)$ for each Set normalizing their performance average by its own average. In the sequence, all Set Managers pass $ISet(j)$ index to the BSP processes under their jurisdiction.

$$Comp(i,j) = P_{comp}(i)\ .\ CTP_{k+\alpha-1}(i)\ .\ ISet_{k+\alpha-1}(j) \tag{3}$$

Equation 3 shows the function to calculate the Computation metric for process i to Set j. The value of the equation 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.

3.3.2 Communication Metric

Communication metric is expressed through $Comm(i,j)$, where i denotes a BSP process and j means the target Set. This metric treats the communication (just receiving actions) involving the process i and all processes that belong to Set j. In order to compute $Comm(i,j)$, data collected at superstep k up to $k+\alpha-1$ is used. Besides this, 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 number of bytes involved during receptions performed by process i from sendings executed by processes that belong to Set j at superstep t. $PB_t(i,j)$ is based on the 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. Figure 6 presents the algorithm which uses this prediction to compute $P_{comm}(i,j)$. This algorithm uses a variable β which informs the acceptable variation in communication prediction. Similarly to δ, if β is 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. $P_{comm}(i,j)$ is the first element in function $Comm(i,j)$. 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, $CommTP_{k+\alpha-1}(i,j)$ is achieved as follows.

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

1. **for** t from superstep k to superstep $k + \alpha - 1$ **do**
2. **if** $(1 - \beta).B_k(i,j) \leq PB_k(i,j)$ and $(1 + \beta).B_k(i,j) \geq PB_k(i,j)$ **then**
3. Increases $P_{comm}(i,j)$ by $\frac{1}{\alpha}$ up to 1
4. **else**
5. Decreases $P_{comm}(i,j)$ by $\frac{1}{\alpha}$ down to 0
6. **endif**
7. **endfor**

Fig. 6. Communication Pattern $P_{comm}(i,j)$

$$Comm(i,j) = P_{comm}(i,j)\ .\ BTP_{k+\alpha-1} \tag{4}$$

The function that computes Communication metric is presented in Equation 4. The result of Equation 4 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.

3.3.3 Memory Metric

Function $Mem(i,j)$ represents the Memory metric and evaluates the migration cost of 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 α is achieved). 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

process i is established with the Set Manager of each considered Set. Finally, the time spent on migration operations of process i to Set j is calculated through $Mig(i,j)$ function. These operations are dependent of operating system, as well as the tool used to offer process 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. However, $Mig(i,j)$ does not depend on the load of Set j.

$$Mem(i,j) = M(i) \,.\, T(i,j) \; + \; Mig(i,j) \tag{5}$$

Equation 5 shows the elements of $Mem(i,j)$. 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 is the transferring of process i to Set j. On the other hand, as $Mem(i,j)$ increases, the migration cost of the process i to Set j increases as well.

3.3.4 Potential of Migration Analysis

We used the notion of force from Physics to create the Potential of Migration (PM) of each process. 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 process. Vectors $\vec{Comp}$ and $\vec{Comm}$ represent the Computation and Communication metrics, respectively. Both have the same direction and stimulate the process migration. On the other hand, the Memory metric means the migration costs and is symbolized by vector $\vec{Mem}$. $\vec{Mem}$ works against the migration, since its direction is opposite to $\vec{Comp}$ and $\vec{Comm}$.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{6}$$

$\vec{Comp}$, $\vec{Comm}$ and $\vec{Mem}$ vectors are combined to create the resultant vector called $\vec{PM}$ (Potential of Migration). Then, $\vec{PM}$ means the resultant force that will decide if a process is a candidate for migration or not. Considering MigBSP context, $\vec{PM}$ will be denoted by $PM(i,j)$ function where i means a process while j represents a specific Set (see Equation 6). Thus, Figure 7 shows the actuation of Computation, Communication and Memory metrics to compute PM. $Comp(i,j)$, $Comm(i,j)$ and $Mem(i,j)$ represent the Computation, Communication and Memory metrics, respectively. The greater the value of $PM(i,j)$, the more prone the 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 costs to j.

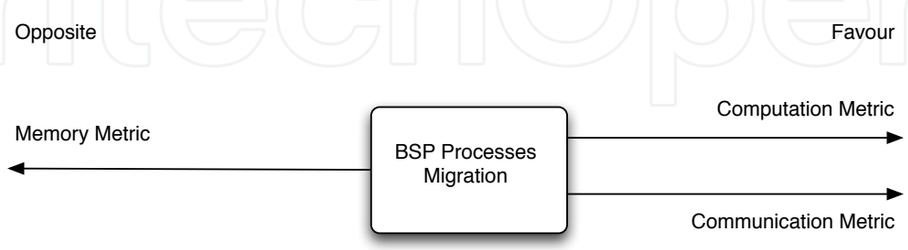


Fig. 7. Resultant force (Potential of Migration): (i) Computation and Communication metrics act in favor of migration; (ii) Memory works in the opposite direction as migration costs

Each process i will compute n times Equation 6, 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. Concerning this, 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. MigBSP uses this list to apply one of two possible heuristics to select the candidates 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 takes one process, the first of the list, whose has the highest PM.

3.4 Analyzing Destination of Elected Processes

Process migration happens after the barrier synchronization of the superstep in which α is reached (see subsection 3.2). An elected process i has a target Set j informed on its Potential of Migration $PM(i,j)$. Thus, the pertinent question is to select which node/processor of this Set can be the destination of the 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.

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 7. $time(p)$ captures the computation power of processor p taking into account the external load (processes that do not belong to the BSP application). $load(p)$ represents the CPU load average on the last 15 minutes. This time interval was adopted based on work of (Moreno-Vozmediano & Alonso-Conde, 2005). Equation 7 also works with instruction summing of each BSP process assigned to processor p in the last executed superstep. In this context, $S(i,p)$ is equal to 1 if a 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 8 based on data from process i, as well as from its own Set.

$$time(p) = \frac{\sum_{i,p:S(i,p)=1} I_{k+\alpha-1}(i)}{(1 - load(p)) \,.\, cpu(p)} \tag{7}$$

$$t1 = time(p) + B_{k+\alpha-1}(i,j) \,.\, T(i,j) \; + \; Mem(i,j) \tag{8}$$

$$t2 = time(p') + B_{k+\alpha-1}(i,j) \,.\, T(i,j) \tag{9}$$

The idea of Equation 8 is to simulate the execution of the 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 cost (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. This last Set Manager computes Equation 9. This equation is used to analyze the execution of process i considering its current execution. In this situation, p' is the current processor of process i and $T(i,j)$ means the transfer rate between the Set of process i and Set j. On both Equations 8 and 9, $B_{k+\alpha-1(i,j)}$ is the amount 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 if $t1 \; < \; t2$.

4. Evaluation Methodology

The main objective of this evaluation is to observe the changes on performance when MigBSP controls the process relocation in different scientific applications. Concerning this, we are testing MigBSP with three applications, which are listed below.

(i) Lattice Boltzmann application - It is widely used in the computational fluid dynamics area. 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 the 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.

While the first application is regular, the other two present an irregular behavior. The regularity issue treats the processes' activities at each superstep. The behaviors of the 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. The evaluation comprises the simulation of the applications on three scenarios.

- Scenario (*i*): Application execution simply;
- Scenario (*ii*): Application execution with scheduler without applying migrations;
- Scenario (*iii*): Application execution with scheduler allowing migrations.

Scenario *ii* consists in performing all scheduling calculus about which processes will migrate, but it does not comprise any migrations. Scenario *iii* enables migrations and adds the migrations costs on those processes that migrate from one processor to another. The difference between scenarios *ii* and *i* represents exactly the overhead imposed by MigBSP. We are using the **SimGrid Simulator** (Casanova et al., 2008) (MSG module), which makes possible application modeling and process migration. This simulator is deterministic, where a specific input always results in the same output. In addition, a time equal to $Mem(i, j)$ is paid for each migration of process i to Set j (see subsection 3.3). We assembled an infrastructure with five Sets, which is depicted in Figure 8. This infrastructure represents the clusters and the network connections that we have at UFRGS University, Brazil. Each node has a single processor. For the sake of simplicity, we hide the network of each cluster. Clusters Labtec, Corisco and Frontal have their nodes linked by Fast Ethernet, while ICE and Aquario use Gigabit connection. The migration costs are based on executions with AMPI (Huang et al., 2006) on our clusters.
Figure 8 also reveals the initial processes-recourses mappings. 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. The notation {(p,m)} is used in the following sections and means that process p is running over machine m (or p will migrate to m). Finally, the tests were executed using α equal to 2, 4, 8 and 16. Furthermore, we employed ω equal to 3 and initial D equal to 0.5. The first application used the heuristic two to select the process for migration, while the other two employ the heuristic one to choose the candidates with x equal to 80%.

5. Results Remarks and Discussions

This section is divided in three subsections, which explain in details the results of each tested application separately. The overall analysis of the results will be presented in the last section.

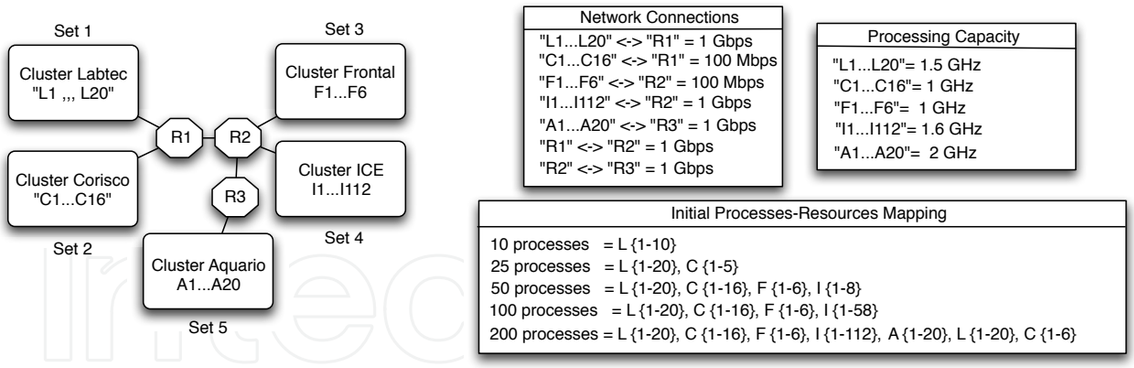


Fig. 8. Testbed infrastructure and the initial processes-resources mappings

5.1 Lattice Boltzmann Method

This method is a powerful technique for the computational modeling of a wide variety of complex fluid flow problems (Schepke, 2007). It models the fluid consisting of particles whose perform consecutive propagation and collision processes over a discrete lattice mesh.

5.1.1 Modeling de Problem

We modeled a BSP implementation of a 2D-based Lattice Boltzmann Method to SimGrid using vertical domain decomposition. 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. At the end of each iteration, data that lie on the boundaries between blocks are passed between the appropriate processes and the superstep is completed. An abstract view of the problem is illustrated in Figure 9.

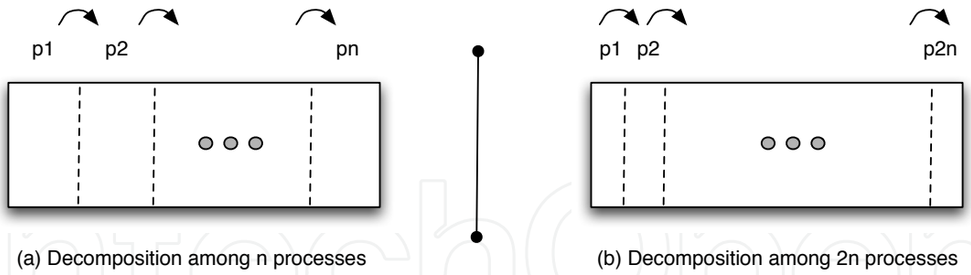


Fig. 9. Different matrix partition organizations when varying the number of processes

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. As we can observe in Figure 9, matrix partition will influence the number of instructions to be executed per process and, consequently, the size of each process 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.

5.1.2 Results and Discussions

Table 1 presents the times when testing 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 α at each call for process 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 obtained negative values of PM when the rescheduling was tested. This fact resulted in an empty list of migration candidates.

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 1. Evaluating 10 processes on three considered scenarios (time in seconds)

The results of the execution of 25 processes are presented in Table 2. In this context, the system remains stable and α grows at each rescheduling call. One migration occurred {(p21,a1)} when testing 10 supersteps and using α equal to 4. Our notation informs that process p21 was reassigned to run on node a1. A second and a third migrations happened when considering 50 supersteps: {(p22,a2), (p23,a3)}. They happened in the next two calls for process rescheduling (at supersteps 12 and 28). When evaluating 2000 supersteps and maintaining this value of α, 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). The first five migrations moved processes from cluster Corisco to Aquario. After that, three processes from Labtec were chosen for migration. Concluding, we obtained a profit of 14% after executing 2000 supersteps when α equal to 4 is used.

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 2. Evaluating 25 processes on three considered scenarios (time in seconds)

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 process 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 10 illustrates the frequency of process 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).

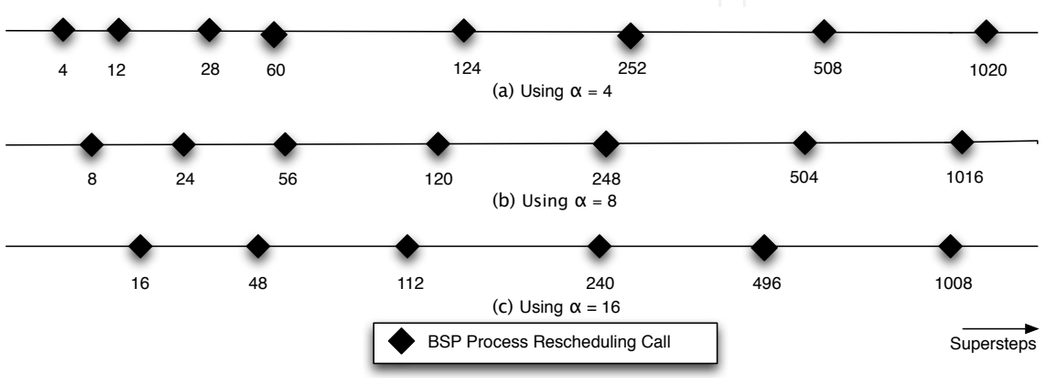


Fig. 10. Number of rescheduling calls when 25 processes and 2000 supersteps are evaluated

Table 3 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 10). 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 4 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 rescheduling call. 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 50 and 100 processes.
The results of scenarios *i*, *ii* and *iii* with 200 processes is shown in Table 5. 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 process 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.

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 3. 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	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

Table 4. 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	1.04	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

Table 5. Evaluating 200 processes on three considered scenarios (time in seconds)

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 Subsection 3.2 for details).

Processes	Scenario *i* - Without process migration	Scenario *iii* - With process 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

Table 6. Barrier times on two situations

Table 6 presents the barrier times captured when 2000 supersteps were tested. More 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 6 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 execution of 50 processes.

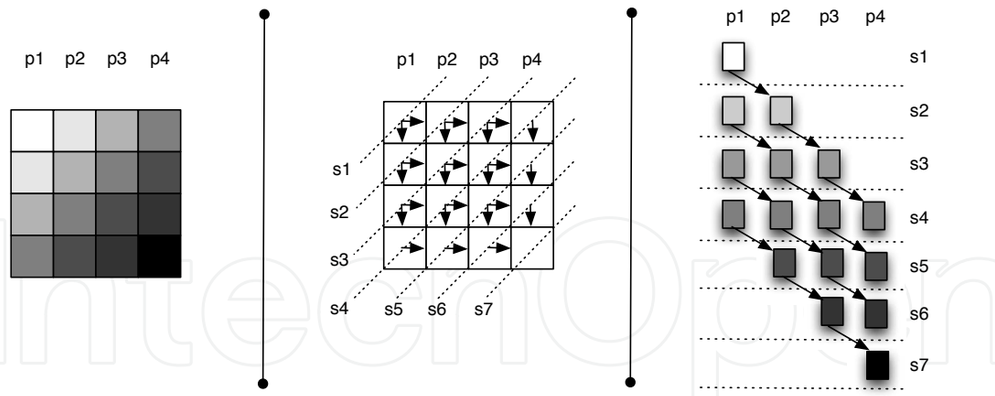
5.2 Smith-Waterman Application

Our second application is based on dynamic programming (DP), which is a popular algorithm design technique for optimization problems (Low et al., 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. $2D/1D$ algorithms are all irregular with changes on load computation density along the matrix's cells. In particular, we observed the Smith-Waterman algorithm that is a well-known 2D/1D algorithm for local sequence alignment (Smith, 1988).

5.2.1 Modeling the Problem

Smith-Waterman algorithm proceeds in a series of wavefronts diagonally across the matrix. Figure 11 (a) illustrates the concept of the algorithm for a 4×4 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 11 (b) shows a 4×4 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: (i) $2n-1$ supersteps are crossed to compute a square matrix with order n and; (ii) each process will be involved on n supersteps. Figures 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 (see Subsection 3.3 for more details) . $P_{comp}(i)$ 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 δ 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 δ 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 10×10 matrix is tested (see details about the notations in Subsection 3.3). The percentage of 50% enforces instruction regularity in the system. Both values of δ 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



(a) Computational load density along the matrix

(b) Mapping of supersteps and communications among the cells

(c) Emphasizing that the load is uniform inside a particular superstep

Fig. 11. Different views of Smith-Waterman irregular application

happen on this scenario. We tested the behavior of square matrixes of order 10, 25, 50, 100 and 200. Each cell of a 10×10 matrix needs to communicate 500 Kbytes and each process occupies 1.2 Mbyte in memory (700 Kbytes comprise other application data). The cell of 25×25 matrix communicates 200 Kbytes and each process occupies 900 Kbytes in memory and so on.

5.2.2 Results and Discussions

Table 7 presents the application evaluation. Nineteen supersteps were crossed when a 10×10 matrix was tested. Adopting this size of matrix and α 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 process 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*, 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 less rescheduling calls and; (ii) the call that causes process migration was done at a specific superstep in which MigBSP takes better decisions.
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 process 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 considering 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

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

Table 7. Evaluation of scenarios *i*, *ii* and *iii* when varying the matrix size

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 the supersteps that include them. Both are executing on a slower cluster and the barrier waits for the slowest process. Maintaining the matrix size and adopting α 16, we have two calls: at supersteps 16 and 48. This last call migrates p24 an p25 to cluster Aquario. Although this movement is pertinent to get performance, just one superstep is executed before ending the application.
Fifty processes were evaluated when the 50×50 matrix was considered. In this context, α also increases at each call for process 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 process rescheduling were done (at supersteps 2, 6, 14, 30 and 62). No migrations are indicated in the first three calls. The greater is the matrix size, the greater is the amount 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 4 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 rescheduling call is done at superstep 60. The same processes were migrated at this point. However, there are two more supersteps to execute using α equal to 4. Three rescheduling calls were done with α8 (at supersteps 8, 24 and 56). Only 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.
The execution with a 100×100 matrix shows good results with process migration. Six rescheduling calls were done when using α 2. Migrations did not occur at the first three supersteps (2, 6 and 14). Process 21 up to 29 are migrated to cluster Aquario after superstep 30. In addition, process 37 to 42 are migrated to cluster Aquario at superstep 62. Finally, superstep 126 indicates 7 migrations, 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 slowest clusters on our infrastructure testbed. α equal to 16 produced 3 attempts for migration when a 100×100 matrix is evaluated (at supersteps 16, 48 and 112). All of them triggered migrations. In the first

call, the 11^{th} 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 after crossing 112 supersteps. They are: p21 to p36. This spectrum of candidates is equal to the processes that are running on Frontal. Considering this, only 3 processes were migrated actually: {(p34,a18),(p35a19),(p36,a20)}.

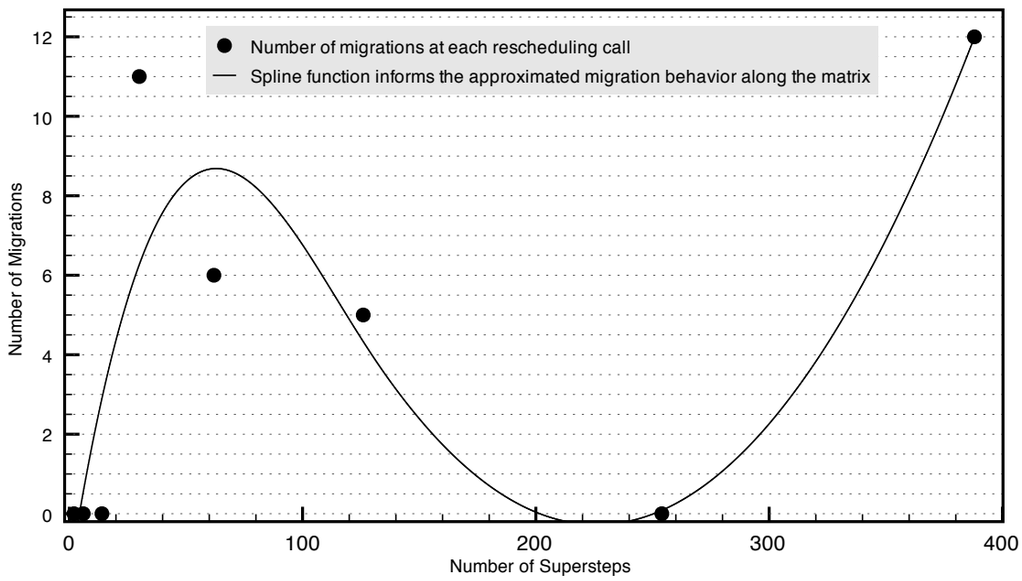


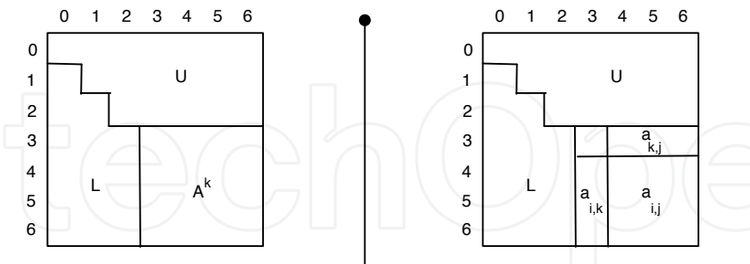
Fig. 12. Migration behavior when testing a 200 × 200 matrix with initial α equal to 2

Table 7 also shows the application performance when the 200×200 matrix was tested. Satisfactory results were obtained with process 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 12 illustrates the migrations behavior along the execution with α 2. Using α 2 and considering scenario *iii*, 8 calls for process rescheduling were done. Migrations were not done at supersteps 2, 6 and 14. 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. Six 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. Finally, 12 processes (p189 to p200) are migrated to cluster Aquario when superstep 388 was crossed. At this time, all previous processes allocated to Aquario are inactive and the migrations are viable. However, just 10 remaining supersteps are executed to amortize the process migration costs.

5.3 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 the LU Decomposition technique. It 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 $0 \leq j < i < n$.



(a) LU decomposition of a 7x7 matrix at the start of stage k=3

(b) Both $a_{i,k}$ and $a_{k,j}$ represent the data dependecy to compute $a_{i,j}$

Fig. 13. L and U matrices with the same memory space of the original matrix A^0

1. **for** k from 0 to $n-1$ **do**
2. **for** j from k to $n-1$ **do**
3. $u_{k,j} = a^k_{k,j}$
4. **endfor**
5. **for** i from $k+1$ to $n-1$ **do**
6. $l^k_{i,k} = \frac{a^k_{i,k}}{u_{k,k}}$
7. **endfor**
8. **for** i from $k+1$ to $n-1$ **do**
9. **for** j from $k+1$ to $n-1$ **do**
10. $a^{k+1}_{i,j} = a^k_{i,j} - l_{i,k} \,.\, u_{k,j}$
11. **endfor**
12. **endfor**
13. **endfor**

for k from 0 to $n-1$ **do**
for i from $k+1$ to $n-1$ **do**
$a_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
endfor
for i from $k+1$ to $n-1$ **do**
for j from $k+1$ to $n-1$ **do**
$a_{i,j} = a_{i,j} - a_{i,k} \,.\, a_{k,j}$
endfor
endfor
endfor

Fig. 14. Two algorithms to solve the LU Decomposition problem

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 13 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. Figure 14 presents the 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. Figure 14 also presents in the right side the functioning of the previous algorithm using just the elements from matrix A. Figure 13 (b) presents the data that is necessary to compute $a_{i,j}$. Besides its own value, $a_{i,j}$ is updated using a value from the same line and another from the same column.

5.3.1 Modeling the Problem

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 $A(i, k+1 : n-1)$ 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 change of the elements in $A(k+1 : n-1, j)$ uses only one value from row k of A, namely $a_{k,j}$. If we divide each column over a set of M processes, the communication of an element of row k can be restricted to a multicast to M processes.
We are using a Cartesian scheme for the distribution of matrices. The square cyclic distribution is used since it is particularly suitable for matrix computations (Bisseling, 2004). Thus, 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 15 depicts a 6×6 matrix mapped to 6 processes, where $M = 2$ and $N = 3$. Assuming that M and N are factors of n, each process will store nc (number of cells) cells in memory (see Equation 10).

$$nc = \frac{n}{M}.\frac{n}{N} \tag{10}$$

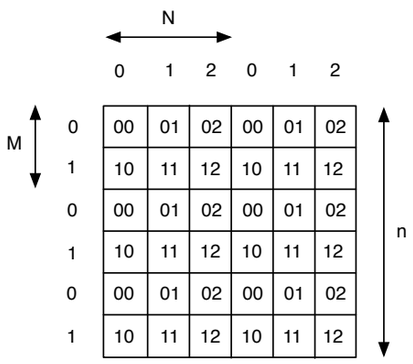


Fig. 15. Cartesian distribution of a matrix over 2×3 ($M \times N$) processes

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 8. 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 application will execute one superstep of type 1 and will follow with the interleaving of supersteps 2 and 3. Thus, a $n \times n$ matrix will trigger $2n+1$ supersteps in our LU modeling. We

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 $a_{i,k}$ $(k+1 \leq i < n)$

Table 8. Modeling three types of supersteps for LU computation

modeled the Cartesian distribution $M \times N$ in the following manner: 5×5, 10×5, 10×10 and 20×10 for 25, 50, 100 and 200 processes, respectively. Moreover, we are applying simulation over square matrices with orders 500, 1000, 2000 and 5000. Lastly, the tests were executed using $\alpha = 4$, $\omega = 3$, $D = 0.5$ and x = 80%.

5.3.2 Results and Discussions

Table 9 presents the results when evaluating LU application. The tests with the first matrix size show the worst results. Formerly, the higher the number of processes, the worse the performance, as we can observe in scenario *i*. The reasons for the observed times are the overheads related to communication and synchronization. Secondly, MigBSP indicated that all migration attempts were not viable due to low computing and communication loads when compared to migration costs. Considering this, both scenarios *ii* and *iii* have the same results.

Processes	500×500 matrix			1000×1000 matrix			2000×2000 matrix		
	i	*ii*	*iii*	*i*	*ii*	*iii*	*i*	*ii*	*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

Table 9. First results when executing LU linked to MigBSP (time in seconds)

When testing a 1000×1000 matrix with 25 processes, the first rescheduling call does not cause migrations. 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. α always increases its value at each rescheduling call since the processes are balanced after the mentioned relocations. MigBSP obtained a gain of 12% of performance with 25 processes when comparing scenarios *i* and *iii*. With the same size of matrix and 50 processes, 6 processes from Frontal were migrated to Aquario at superstep 9. Although these migrations are profitable,

they do not provide stability to the system and the processes remain unbalanced among the resources. Migrations are not viable in the next 3 calls at supersteps 15, 21 and 27. After that, MigBSP launches our second adaptation on rescheduling frequency in order to alleviate its impact and α begins to grow until the end of the application. The tests with 50 processes obtained gains of just 5% with process migration. This is explained by the fact that the computational load is decreased in this configuration when compared to the one with 25 processes. In addition, the bigger the number of the superstep, the smaller the computational load required by it. Therefore, the more advanced the execution, the lesser the gain with migrations. The tests with 100 and 200 processes do not present migrations owing to the forces that act in favor of migration are weaker than the Memory metric in all rescheduling calls.
The execution with a 2000×2000 matrix presents good results because the computational load is increased. We observed a gain of 15% with process 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 more computations. The time for concluding the 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. However, MigBSP indicated the migration of 14 processes, since there were only 14 unoccupied processors in the target cluster.

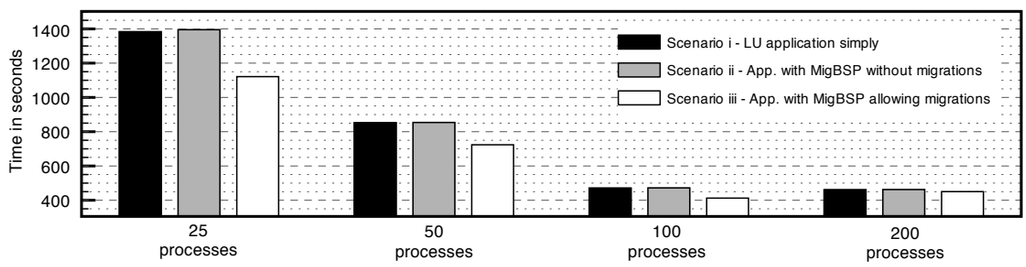


Fig. 16. Performance graph with our three scenarios for a 5000×5000 matrix

We observed that the higher the matrix order, the better the results with process migration. Considering this, the evaluation of a 5000×5000 matrix can be seen in the Figure 16. The simple movement of all processes 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. The same migration behavior found on the tests with the 2000×2000 matrix was achieved in Scenario *iii* 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 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. Thus, the nodes that belong to Corisco just execute one BSP process while the nodes from Aquario begin to treat 2 processes. The remaining rescheduling calls inform the processes from Labtec as those with the higher values of PM. However, their migrations are not considered profitable. The final execution with 200 processes achieved 460.85s and 450.33s for scenarios *i* and *iii*, respectively.

6. Conclusion

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 job. In the second level processes from a job are (re)scheduled using this pool of processors. MigBSP can be included in this last scheme, offering algorithms for load (BSP processes) rebalancing among the resources during the application runtime. In the best of our knowledge, MigBSP is the pioneer model on treating BSP process rescheduling with three metrics and adaptations on remapping frequency. These features are enabled by MigBSP at middleware level, without changing the application code.

Considering the spectrum of the three tested 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. Considering this last topic, MigBSP indicates migrations even when the application is close to finish. In this situation, these migrations bring an overhead since the remaining time for application conclusion is too short to amortize their costs.

The results showed that MigBSP presented a low overhead on application execution. The calculus of the PM (Potential of Migration) as well as our efficient adaptations were responsible for this feature. PM considers processes and Sets (different sites), not performing all processes-resources tests at the rescheduling moment. Meanwhile, our adaptations were crucial to enable MigBSP as a viable scheduler. Instead of performing the rescheduling call at each fixed interval, they manage a flexible interval between calls based on the behavior of the processes. The concepts of the adaptations 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 ω calls is observed.

7. References

Bhandarkar, M. A., Brunner, R. & Kale, L. V. (2000). Run-time support for adaptive load balancing, *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, Springer-Verlag, London, UK, pp. 1152–1159.

Bisseling, R. H. (2004). *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, Oxford University Press.

Bonorden, O. (2007). Load balancing in the bulk-synchronous-parallel setting using process migrations., *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, IEEE, pp. 1–9.

Bonorden, O., Gehweiler, J. & auf der Heide, F. M. (2005). Load balancing strategies in a web computing environment, *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, Poznan, Poland, pp. 839–846.

Casanova, H., Legrand, A. & Quinson, M. (2008). Simgrid: A generic framework for large-scale distributed experiments, *Tenth International Conference on Computer Modeling and Simulation (uksim)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 126–131.

Casavant, T. L. & Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems, *IEEE Trans. Softw. Eng.* **14**(2): 141–154.

Chen, L., Wang, C.-L. & Lau, F. (2008). Process reassignment with reduced migration cost in grid load rebalancing, *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* pp. 1–13.

Du, C., Ghosh, S., Shankar, S. & Sun, X.-H. (2004). A runtime system for autonomic rescheduling of mpi programs, *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing*, IEEE Computer Society, Washington, DC, USA, pp. 4–11.

Du, C., Sun, X.-H. & Wu, M. (2007). Dynamic scheduling with process migration, *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, pp. 92–99.

Frachtenberg, E. & Schwiegelshohn, U. (2008). New Challenges of Parallel Job Scheduling, *Job Scheduling Strategies for Parallel Processing* **4942**: 1–23.

Heiss, H.-U. & Schmitz, M. (1995). Decentralized dynamic load balancing: the particles approach, *Inf. Sci. Inf. Comput. Sci.* **84**(1-2): 115–128.

Hernandez, I. & Cole, M. (2007). Scheduling dags on grids with copying and migration., *in* R. Wyrzykowski, J. Dongarra, K. Karczewski & J. Wasniewski (eds), *PPAM*, Vol. 4967 of *Lecture Notes in Computer Science*, Springer, pp. 1019–1028.

Huang, C., Zheng, G., Kalé, L. & Kumar, S. (2006). Performance evaluation of adaptive mpi, *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM Press, New York, NY, USA, pp. 12–21.

Kondo, D., Casanova, H., Wing, E. & Berman, F. (2002). Models and scheduling mechanisms for global computing applications, *IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, Washington, DC, USA, p. 79.2.

Low, M. Y.-H., Liu, W. & Schmidt, B. (2007). A parallel bsp algorithm for irregular dynamic programming, *Advanced Parallel Processing Technologies, 7th International Symposium*, Vol. 4847 of *Lecture Notes in Computer Science*, Springer, pp. 151–160.

Milanés, A., Rodriguez, N. & Schulze, B. (2008). State of the art in heterogeneous strong migration of computations, *Concurr. Comput. : Pract. Exper.* **20**(13): 1485–1508.

Moreno-Vozmediano, R. & Alonso-Conde, A. B. (2005). Influence of grid economic factors on scheduling and migration., *High Performance Computing for Computational Science - VECPAR*, Vol. 3402 of *Lecture Notes in Computer Science*, Springer, pp. 274–287.

Sánchez, A., Pérez, M. S., Montes, J. & Cortes, T. (2010). A high performance suite of data services for grids, *Future Gener. Comput. Syst.* **26**(4): 622–632.

Schepke, Claudio; Maillard, N. (2007). Performance improvement of the parallel lattice boltzmann method through blocked data distributions, *19th International Symposium on Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007*, pp. 71–78.

Smith, J. M. (1988). A survey of process migration mechanisms, *SIGOPS Oper. Syst. Rev.* **22**(3): 28–40.

Tanenbaum, A. (2003). *Computer Networks*, 4th edn, Prentice Hall PTR, Upper Saddle River, New Jersey.

Utrera, G., Corbalan, J. & Labarta, J. (2005). Dynamic load balancing in mpi jobs, *The 6th International Symposium on High Performance Computing*.

Vadhiyar, S. S. & Dongarra, J. J. (2005). Self adaptivity in grid computing: Research articles, *Concurr. Comput. : Pract. Exper.* **17**(2-4): 235–257.

Valiant, L. G. (1990). A bridging model for parallel computation, *Commun. ACM* **33**(8): 103–111.

Future Manufacturing Systems

Edited by Tauseef Aized

ISBN 978-953-307-128-2

Hard cover, 268 pages

Publisher Sciyo

Published online 17, August, 2010

Published in print edition August, 2010

This book is a collection of articles aimed at finding new ways of manufacturing systems developments. The articles included in this volume comprise of current and new directions of manufacturing systems which I believe can lead to the development of more comprehensive and efficient future manufacturing systems. People from diverse background like academia, industry, research and others can take advantage of this volume and can shape future directions of manufacturing systems.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen