Applying Process Migration on a BSP-Based LU Decomposition Application

Rodrigo da Rosa Righi[1], Laércio Lima Pilla[1], Alexandre Carissimi[1], Philippe Olivier Alexandre Navaux[1] and Hans-Ulrich Heiss[2]

[1] Institute of Informatics - Federal University of Rio Grande do Sul - Brazil
[2] Kommunikations- und Betriebssysteme - Technical University Berlin - Germany

Abstract. Process migration is an useful mechanism to offer load balancing. In this context, we developed a model called MigBSP that controls processes rescheduling on BSP applications. MigBSP is especially pertinent to obtain performance on this type of applications, since they are composed by supersteps which always wait for the slowest process. In this paper, we focus on the BSP-based modeling of the widely used LU Decomposition algorithm as well as its execution with MigBSP. The use of multiple metrics to decide migrations and adaptations on rescheduling frequency turn possible gains up to 19% over our cluster-of-clusters architecture. Finally, our final idea is to show the possibility to get performance in LU effortlessly by using novel migration algorithms.

1 Introduction

A possibility to increase performance on dynamic and heterogeneous environments comprises the processes' relocation. Generally, process migration is implemented within the application. This organization results in a close coupling between the application and the algorithms' data structures, which makes this approach non-extensible. Even more, some initiatives use explicit calls in the application code [2]. A different approach for migration happens at middleware level, linking the balancer tool with the programming library directly /[15]. Commonly, this approach does not require changes in the application code nor previous knowledge about the system.

A typical scheduling middleware applies mechanisms to allocate the processes with longer computing times to faster machines. Nevertheless, this approach is not the best one for irregular applications and dynamic distributed environments, because a good process-resource assignment performed in the beginning of the application may not remain efficient with time [1, 12, 14]. At this moment, it is not possible to recognize either the amount of computation of each process nor the communication patterns among them. Besides fluctuations in the processes' computation and communication actions, the processors' load may vary and networks can become congested while the application is running. Therefore, an alternative is to perform process rescheduling through their migration to new resources, offering application runtime load balancing [7, 10].

In this context, we designed a process rescheduling model called **MigBSP** that works over **BSP** (Bulk Synchronous Parallel) applications [18, 8]. It explores the automatic and transparent load (processes) balancing at middleware level. To make decisions about load balancing, the model considers data about the infrastructure, the processes' behavior as well as migration costs. MigBSP was organized to work with BSP applications, once they are based on synchronous phases (supersteps). Thus, the main idea of the model is to reduce the duration of each superstep, decreasing the application time as well. MigBSP contributions are twofold: (*i*) combination of multiple metrics to select the candidates for migration and; (*ii*) minimization of the model's overhead with adaptation that act over the rescheduling frequency.

The BSP model is mainly used for the development of scientific applications such as data mining, sorting and fluid dynamics [4]. Particularly, this paper presents the modeling of a parallel BSP-based version of the widely used **LU Decomposition** method [3]. In addition, it describes the execution of this application when linked to MigBSP over a cluster-of-clusters environment. The LU decomposition splits a matrix A in the product of a lower triangular matrix L and an upper triangular matrix U. LU is employed to turn the calculation of linear equations easier, since the solution of a triangular set of equations is trivial. Besides its usage, we choose LU because some initiatives impose changes in the code and/or extra executions when offering load balancing for this application[2, 11, 17]. Thus, the paper's final idea is to show the possibility for getting performance in LU application effortlessly by using novel migration algorithms.

2 MigBSP: Process Rescheduling Model

MigBSP manages load balancing issues, where the load is represented by BSP processes, aiming to reduce each superstep time of the application. Its key idea is to migrate processes which have a long computation time, perform several communication actions with other processes whose belong to a same site (*e.g.*, a cluster) and present low migration costs. Figure 1 (a) illustrates a superstep k of a BSP application in which the processes are not balanced among the resources. Figure 1 (b) shows the expected result with the rescheduling of the processes after superstep k, which will influence the execution of the next supersteps, (including k+1, k+2 and so on).

MigBSP's architecture is heterogeneous and composed by clusters and supercomputers. This architecture is assembled with the abstractions of Sets (different sites) and Set Managers. Set Managers are responsible for scheduling, capturing data from a specific Set and exchanging it among other managers. MigBSP can be seen as a scheduling middleware. There is no need for changes in the application code. All data necessary for its functioning may be captured directly in both communication and barrier functions as well as in other sources like the operating system. We described the first ideas of MigBSP in [16]. However, such work presents an evaluation with a synthetic application with a reduced number of supersteps (up to 400) and processes (up to 10).

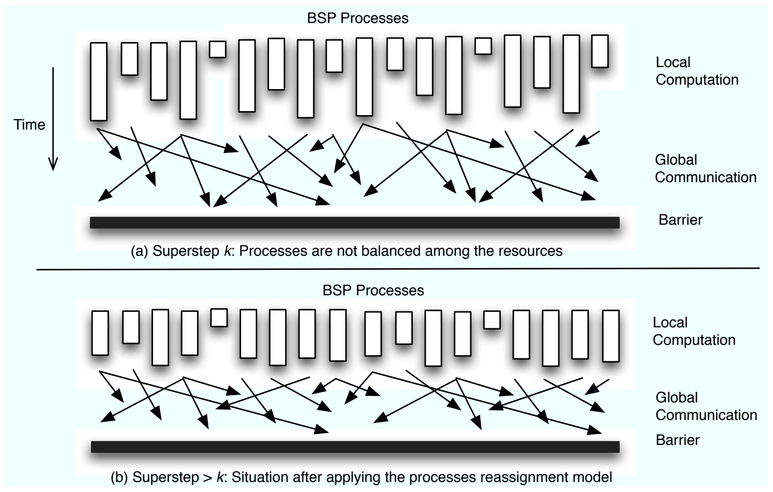

BSP Processes
Local Computation
Time
Global Communication
Barrier
(a) Superstep k: Processes are not balanced among the resources
BSP Processes
Local Computation
Global Communication
Barrier
(b) Superstep > k: Situation after applying the processes reassignment model

Fig. 1. BSP Supersteps in two different situations

The decision for process remapping is taken at the end of a superstep, after the barrier. At this moment, we can analyze data from all BSP processes. Aiming to generate as less intrusiveness in application as possible, we applied two adaptations that control the value of α - the adaptive period between rescheduling calls. 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 ω calls is observed. A variable D is used to indicate a percentage of how far the slowest and the fastest processes may be from the average. Our second adaptation works on increasing D. The higher its value, the greater the odds to increase α.

We employed a decision function called **Potential of Migration** (PM) to select the candidates for migration. Each process i computes q functions $PM(i,j)$, where q is the number of Sets and j means a specific Set. The main idea consists in performing a subset of the processes-resources tests at the rescheduling moment. $PM(i,j)$ is found using the Computation, Communication and Memory metrics (see Equation 1). Computation metric - $Comp(i,j)$ - considers a Computation Pattern $P_{comp}(i)$ that measures the regularity of a process i at its computation phase. This value is close to 1 if the process performs a similar number of instructions at each superstep and close to 0 otherwise. This metric also performs a computation time prediction based on data between rescheduling calls. In the same way, Communication metric - $Comm(i,j)$ - computes the Communication Pattern $P_{comm}(i,j)$ between processes and Sets. Furthermore, it uses a communication time prediction considering data between the rebalancing calls. Memory metric - $Mem(i,j)$ - considers process memory, transferring rate between the process and the manager of target Set, as well as migration costs.

$$PM(i,j) = Comp(i,j) + Comm(i,j) - Mem(i,j) \tag{1}$$

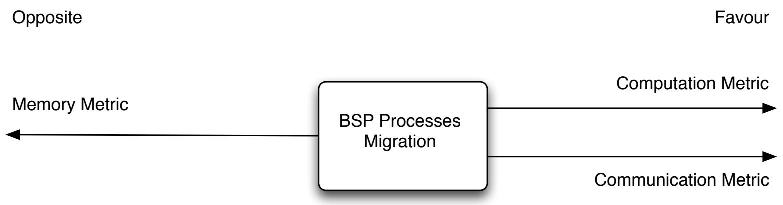


Fig. 2. Operation of the metrics to evaluate the Potential of Migration (PM) of a process: (i) Computation and Communication metrics act in favor of migration; (ii) Memory metric works in the opposite direction as migration costs

Figure 2 depicts the operation of the considered metrics on process migration. Firstly, the BSP processes calculate $PM(i,j)$ locally. At each rescheduling call, each process passes its highest $PM(i,j)$ to its Set Manager which exchanges the PM of its processes among other managers. We used a heuristic to choose the candidates which is based on a decreasing ordered list of PMs. The processes with PM higher than $MAX(PM).x$ are candidates, where x is a percentage. The $PM(i,j)$ of a candidate process i is associated to a Set j. Therefore, the manager of Set j will select the most suitable processor to receive this process. Before a migration, its viability is verified by computing two times: t_l and t_d. t_l means the local execution of process i, while t_d encompasses its prediction of execution on the destination processor and includes the migration costs. For each candidate is chosen a new resource (if $t_l > t_d$) or its migration is canceled.

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. 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 $0 \leq j < i < n$.

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 3 (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 1 presents a sequential algorithm for producing L and U in stages. Stage k first computes the elements

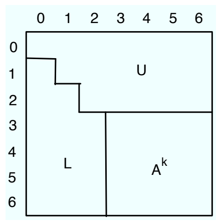


Fig. 3. L and U matrices decomposition using the same memory space of the original matrix A^0

Algorithm 1 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:         $l^k_{i,k} = \frac{a^k_{i,k}}{u_{k,k}}$
 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 2 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:         $a_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
 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
```

$u_{k,j}$, $j \geq k$, 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 2 presents the functioning of the previous algorithm using just the elements from matrix A. Figure 3 (b) presents the data that is necessary to compute $a_{i,j}$ in the last statement of the Algorithm 2. Besides its own value, $a_{i,j}$ is updated using a value from the same line and another from the same column.

4 BSP-based LU Application 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 $A(i, k+1 : n-1)$ 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 [3]. 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 4 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 2).

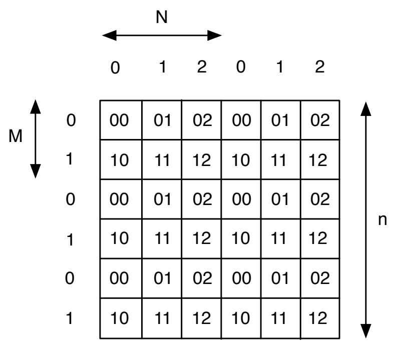


Fig. 4. Cartesian distribution of a 6×6 ($n \times n$) matrix over 2×3 ($M \times N$) processors. The label "*st*" in the cell denotes its owner, process $P(s, t)$

$$nc = \frac{n}{M} . \frac{n}{N} \tag{2}$$

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 [4]. Concerning this, we modeled our algorithm using three kinds of supersteps. They are explained in Table 1. The element $a_{k,k}$ is passed to the process that computes $a_{i,k}$ in the first kind of superstep.

Table 1. Modeling three types of supersteps for LU computation

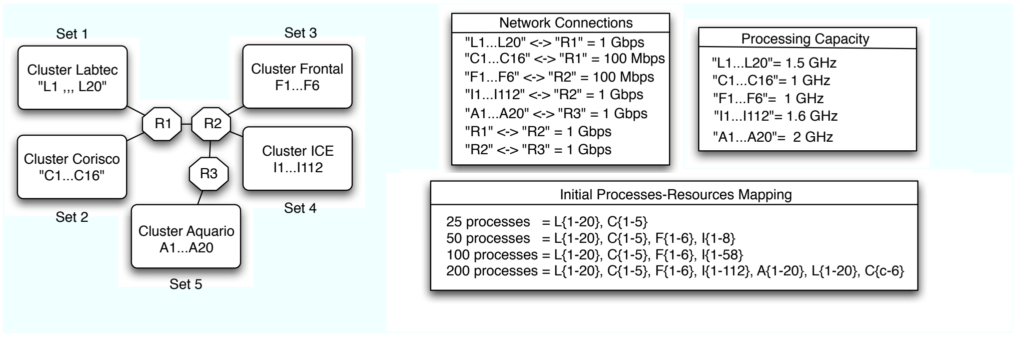
Type of superstep	Steps and explanation
First	Step 1.1 : $k = 0$
	Step 1.2 - Pass the element $a_{k,k}$ to cells which will compute $a_{i,k}$ $(k+1 \leq i < n)$
Second	Step 2.1 : Computation of $a_{i,k}$ $(k+1 \leq i < n)$ by cells which own them
	Step 2.2 : For each i $(k+1 \leq i < n)$, pass the element $a_{i,k}$ 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 $a_{i,k}$ $(k+1 \leq i < n)$

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.

5 Evaluation Methodology

We applied simulation in three scenarios: (*i*) Application execution simply; (*ii*) Application execution with MigBSP without applying migrations; (*iii*) Application with MigBSP allowing migrations. 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 on those processes that migrate from one processor to another. The difference between scenarios ii and i represents exactly the overhead imposed by MigBSP. The analysis of scenarios i and iii will show the final gain or loss of performance when process migration is applied.

We are using the SimGrid Simulator [6] (MSG module), which makes possible application modeling and process migration. This simulator is deterministic, where a specific input always results in the same output. We assembled an infrastructure with five Sets, which is depicted in Figure 5. Each node has a single processor. These Sets are based on a real cluster-of-clusters infrastructure located at Federal University of Rio Grande do Sul, Brazil. Clusters Labtec, Corisco and Frontal have their nodes linked by Fast Ethernet, while ICE and Aquario are clusters with a Gigabit connection. The migration costs are based on real executions with AMPI [12].



Network Connections

"L1...L20" <-> "R1" = 1 Gbps
"C1...C16" <-> "R1" = 100 Mbps
"F1...F6" <-> "R2" = 100 Mbps
"I1...I112" <-> "R2" = 1 Gbps
"A1...A20" <-> "R3" = 1 Gbps
"R1" <-> "R2" = 1 Gbps
"R2" <-> "R3" = 1 Gbps

Processing Capacity

"L1...L20"= 1.5 GHz
"C1...C16"= 1 GHz
"F1...F6"= 1 GHz
"I1...I112"= 1.6 GHz
"A1...A20"= 2 GHz

Initial Processes-Resources Mapping

25 processes = L{1-20}, C{1-5}
50 processes = L{1-20}, C{1-5}, F{1-6}, I{1-8}
100 processes = L{1-20}, C{1-5}, F{1-6}, I{1-58}
200 processes = L{1-20}, C{1-5}, F{1-6}, I{1-112}, A{1-20}, L{1-20}, C{c-6}

Fig. 5. Cluster-of-clusters environment with five Sets and the initial-processes mapping

Figure 5 presents the initial processes-resources mappings for each number of BSP processes. When the number of processes is greater than processors, the mapping begins again from the first Set. We 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 $x = 80\%$.

6 Results Analysis

Table 2 presents the results when evaluating the 500×500, 1000×1000 and 2000×2000 matrices. 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 time results.

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.

Table 2. First results when executing LU linked to MigBSP (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

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 only 14 processes, since there were only 14 unoccupied processors in the target cluster.

The execution of 100 processes presented the same behavior of the execution with 50 processes. Nevertheless, the performance gain was reduced to 4% with 100 processes given the reduction of the workload per process.

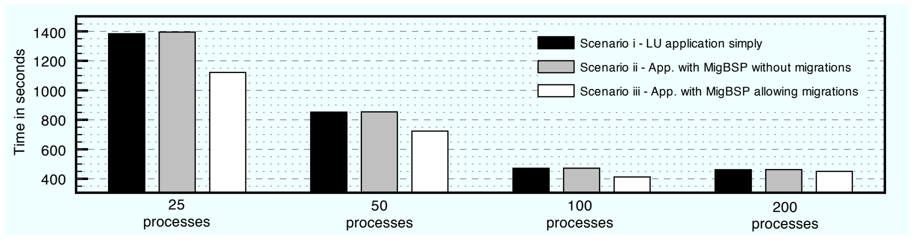


Fig. 6. Performance graph with our three scenarios for a 5000 × 5000 matrix

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 6. 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.

7 Related Work

Bhandarkar, Brunner and Kale presented a support for adaptive load balancing in MPI-based LU application [2]. Periodically, the MPI application transfers control to the load balancer using a special call MPI_Migrate(). Processes reorganization on LU application is proposed by Ennes et al [17]. Such work imposes an extra execution for getting parameters for a communication-graph construction.

Concerning the BSP scope, Jiang, Tong and Zhao presented resource load balancing based on agents [13]. Load balancing is launched when a new task is inserted and it is based on the load rank of the nodes. Scheduling service sends this new task to the current lightest node. Load value is calculated taking such information: CPU, memory resource, number of current tasks and number of network links. In addition, we can cite two works that present migration on BSP applications. The first one describes the PUBWCL library, which exploits the computing cycles of idle computers [5]. PUBWCL can migrate a process during its computation phase and after the barrier. All algorithms just use data about the nodes and consider the computation times from each process.

Other work on BSP context comprises the implementation of the PUB library [4]. The author explains that a load balancer decides when to launch the processes migration, but this issue is not addressed in [4]. He proposed both centralized and distributed strategies for load balancing. In the distributed approach, every node chooses c nodes randomly and asks for their load. One process is migrated if the minimum load found is smaller than the load of its current node. Both strategies take into consideration neither the processes communication nor the migration costs.

8 Concluding Remarks

Scheduling schemes for multi-programmed parallel systems can be viewed in two levels [9, 19]. 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 application runtime. Our model can be seen as a scheduler middleware that does not need changes neither in application code nor knowledge about it and the system infrastructure. Especially, this paper presented MigBSP shortly as well as a modeling and an execution of a BSP-based LU application. The tests when linking it to the LU application enabled us to conclude encouraging results: gains of performance and a short overhead of MigBSP. Contrary to existing works[2, 11, 17], these results are obtained without modifying the application code and without extra executions to feed the load balancing model.

The short overhead of MigBSP is enabled mainly by using efficient adaptations and through the rapid calculus of the scheduling decisions. Firstly, PM (Potential of Migration) 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. Their concepts 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.

For example, the low overhead of MigBSP may be expressed when executing 50 processes and a 2000×2000 matrix. In this context, it adds 3% of costs (MigBSP algorithms are enabled but no migrations are performed). Formerly, this feature is due to the simplicity of the PM evaluation, since it considers the hierarchy notion and employs heuristics. Secondly, MigBSP adaptations work to turn the model viable, especially when migrations cause performance gains but the system remains unbalanced. This occurred with a matrix of order 1000 and 50 processes. Besides this, we observed that the larger the matrix size, the bigger the gain with migrations. Thus, MigBSP obtained the best results with a 5000×5000 matrix. In this situation, we can observe gains larger then 15% when applying our migrations decisions on application execution. Gains of 19% and 15% were obtained when running 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.

Acknowledgements

This work was funded by the following Brazilian Agencies: CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

References

1. G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–265, New York, NY, USA, 2003. ACM Press.
2. M. A. Bhandarkar, R. Brunner, and L. V. Kale. Run-time support for adaptive load balancing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1152–1159, London, UK, 2000. Springer-Verlag.
3. R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI.* Oxford University Press, 2004.
4. O. Bonorden. Load balancing in the bulk-synchronous-parallel setting using process migrations. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–9. IEEE, 2007.
5. O. Bonorden, J. Gehweiler, and F. M. auf der Heide. Load balancing strategies in a web computing environment. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 839–846, Poznan, Poland, September 2005.
6. H. Casanova, A. Legrand, and M. Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Tenth International Conference on Computer Modeling and Simulation (uksim)*, pages 126–131, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
7. L. Chen, C.-L. Wang, and F. Lau. Process reassignment with reduced migration cost in grid load rebalancing. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–13, April 2008.
8. R. da Rosa Righi, L. L. Pilla, A. Carissimi, P. Navaux, and H.-U. Heiss. Migbsp: A novel migration model for bulk-synchronous parallel processes rescheduling. *High Performance Computing and Communications, 10th IEEE International Conferenc e on*, 0:585–590, 2009.
9. E. Frachtenberg and U. Schwiegelshohn. New Challenges of Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, (4942):1–23, May 2008.
10. I. Galindo, F. Almeida, and J. M. Badía-Contelles. 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, volume 5205 of *Lecture Notes in Computer Science*, pages 64–74. Springer, 2008.

11. F. G. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev.*, 47(1):31–55, 2003.
12. C. Huang, G. Zheng, L. Kale, and S. Kumar. Performance evaluation of adaptive mpi. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21, New York, NY, USA, 2006. ACM Press.
13. Y. Jiang, W. Tong, and W. Zhao. Resource load balancing based on multi-agent in servicebsp model. In *International Conference on Computational Science (3)*, volume 4489 of *Lecture Notes in Computer Science*, pages 42–49. Springer, 2007.
14. M. Y.-H. Low, W. Liu, and B. Schmidt. A parallel bsp algorithm for irregular dynamic programming. In *Advanced Parallel Processing Technologies, 7th International Symposium*, volume 4847 of *Lecture Notes in Computer Science*, pages 151–160. Springer, 2007.
15. J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep, and M. den Burger. Middleware adaptation with the delphoi service. *Concurrency and Computation: Practice & Experience*, 2006.
16. R. Righi, L. Pilla, A. Carissimi, and P. O. A. Navaux. Controlling processes reassignment in bsp applications. In *20th International Symposium on Computer Architecture and high Performance Computing (SBAC-PAD 2008)*, pages 37–44. IEEE Computer Society, 2008.
17. R. E. Silva, G. Pezzi, N. Maillard, and T. Diverio. Automatic data-flow graph generation of mpi programs. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 93–100, Washington, DC, USA, 2005. IEEE Computer Society.
18. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
19. M. Wieczorek, S. Podlipnig, R. Prodan, and T. Fahringer. Bi-criteria scheduling of scientific workflows for the grid. *ccgrid*, 0:9–16, 2008.