JUMP: A Unified Scheduling Policy with Process Migration

Juliano F. Ravasi[1], Marcos J. Santana[1], Regina Helena C. Santana[1]

[1]Inst. de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brazil

{jravasi,mjs,rcs}@icmc.usp.br

Abstract. *This paper presents the project and implementation of JUMP, a scheduling policy with process migration support. This new policy unifies initial deployment and process migration in a single algorithm, allowing decision sharing for the common goal of providing a better performance for CPU-bound applications in heterogeneous clusters. The policy is implemented over the dynamical and flexible environment AMIGO, adapted in order to support process migration. Performance evaluation showed that the new policy offers expressive gains in response times when compared to other two scheduling policies implemented in AMIGO in almost all scenarios, for different applications and different environment load situations.*

1. Introduction

Distributed computing systems have been widely used as a means for high-performance computing, allowing the distributed execution of parallel applications, only possible before by means of superscalar, multiprocessor and multicomputer architectures. Distributed systems are attractive for presenting lower costs, providing greater flexibility and being simpler to build and maintain than specialized architectures.

In order to execute a parallel application in a distributed environment, a global scheduling policy must be used. This policy operates one or more algorithms that distribute parallel tasks among the available nodes of the distributed system.

Generally, global scheduling policies try to determine the processing element that will execute the process from its creation until its termination. Such policies are called initial deployment policies and do not perform preemption. The main objective of such policies is to determine which processing elements will be receivers of new tasks, in order to achieve load sharing or load balancing through the distributed system [Shivaratri et al. 1992].

However, for a number of reasons, the scheduling policy may not be able to continually sustain a good load balancing using only initial deployment. The lack of information about the distributed system or the application being scheduled, or the impossibility to predict future changes in the workload of the processing nodes caused by events external to the environment upon which the parallel application is being executed, may cause the system to have degraded performance. In these situations, it is desirable to move (or migrate) processes during their execution between different processing elements of the distributed system in order to achieve a better load distribution.

This paper approaches process migration as a means of enabling more efficient load balancing in distributed parallel environments, through a new global process scheduling policy that unifies the decisions of initial deployment and process migration. A brief review of the concepts related to process migration is presented, followed by a description of the flexible and dynamic scheduling environment AMIGO [Souza et al. 1999], upon which the policy was implemented and evaluated. The project and the implementation of the scheduling policy JUMP are presented, followed by the results obtained through its performance evaluation. Finally, the conclusions are presented.

2. Process Migration

The migration of a process consists in the transfer of a significant subset of its state between two computers of a distributed system, in order to continue the execution in the destination node from the point it was in the originating node [Milojičić et al. 2000]. The migrated process resumes its processing in the destination node with the same state it had in the originating node, thus preserving any processing already performed. Process migration is a particular case of code mobility [Carzaniga et al. 1997][Fuggetta et al. 1998].

In the context of distributed systems, process migration is necessary when the system is found in a state where the distribution of the tasks among the nodes is inefficient or undesirable, generally degrading its performance. Process migration allows modifying the distribution of tasks among the nodes in order to achieve a certain goal. Four common goals for process migration can be listed: dynamic load balancing, approximating processes from resources accessed, fault tolerance and system administration.

Load balancing is, probably, the main goal of process migration and also where there is a greater amount of research. Even using scheduling policies capable of doing load balancing during initial deployment, after some time the system may get to a state of unbalance in the load distribution among the nodes. In this case, processes from more loaded nodes are migrated to less loaded nodes, making easy the equalization of the work load of the distributed parallel environment [Milojičić et al. 2000].

The second goal of process migration is to allow processes to migrate to nodes that are nearer or have easier access to data or resources that they use [Milojičić et al. 2000][Gray et al. 2002][Noguchi et al. 2008]. This is useful in heterogeneous distributed systems, where some nodes have physical resources that cannot be moved (e.g.: memory, terminal), or which the cost of its access or its transfer through the network is greater than the migration of the process (e.g.: a big database).

The third goal, fault tolerance, is possible by detecting partial system failures, allowing processes to be migrated to other nodes before the general fault, while the system is still capable of migration [Sankaran et al. 2005][Wang et al. 2008].

Finally, process migration also allows better system administration, since computers can be either powered off or restarted for maintenance at any time without losing running processes [Milojičić et al. 2000].

Two components determine how process migration is performed: its policy and its mechanism. Generally, these algorithms are implemented independently and in distinct levels. The process migration mechanism is implemented in a lower level of the platform, sometimes integrated with the operating system kernel, while the policy is implemented in a higher level, as a user process.

A process migration policy is basically a load distribution policy, which is composed by four components [Shivaratri et al. 1992]: transfer policy, selection policy, location policy and information policy. The information policy can be implemented as a separate module (called load information management), which is responsible for collecting and sharing load information among the nodes of the distributed system [Milojičić et al. 2000]. Thus, the process migration policy is implemented in only three components: the transfer (or activation) policy, which determines when process migration should happen; the selection policy, which determines the processes that should be migrated; and the location policy, which determines the destination of the processes that are to be migrated [Milojičić et al. 2000].

As soon as it is decided that process migration should happen and it is already known which nodes will be partners (chosen by the migration policy), the migration mechanism is activated in order to execute the transfer itself. The migration mechanism (sometimes called migration algorithm) is responsible for suspending the execution of the process in its originating node, transfer its state to the destination node and restarting its execution in the destination node. These basic tasks are split in a number of steps, which vary in form and order for each migration mechanism.

The process migration mechanism may implement one among several existing algorithms, known as: *eager copy*, *lazy copy*, *pre-copy*, *post-copy* and *flushing*. Descriptions of these algorithms are found in [Milojičić et al. 2000][Richmond and Hitchens 1997][Douglis and Ousterhout 1997].

3. The AMIGO Scheduling Environment

AMIGO (*dyn**AM**ical flex**I**ble schedulin**G** envir**O**nment*) is a flexible and dynamic software environment responsible for process scheduling management in distributed systems, offering an interface through which message-passing environments or distributed applications may request scheduling services. It is an open software tool, developed by the Distributed Systems and Concurrent Programming Group of the Institute of Mathematical Sciences and Computing during Souza's doctoral thesis [Souza et al. 1999] and complemented by several other research works [Araújo et al. 1999][Figueiredo et al. 2002][Santos et al. 2001][Campos Jr. 2001][Voorsluys 2006] developed in the same research group.

Using AMIGO, different scheduling policies can be implemented and independently executed, each one serving a distinct group of distributed applications, which can be configured either by the user or by the system administrator.

The scheduling environment provided by AMIGO consists of three modules: the core daemon, the scheduling policies and the clients that request scheduling. The core is the server process *amigod* (*AMIGO Daemon*), which acts as a bridge between the clients and the scheduling policies. Clients are usually message-passing environments such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) modified to use AMIGO to perform scheduling, but may also be the distributed applications themselves.

Process scheduling happens in two distinct situations. The first occurs when the user or an already-running parallel application requests the creation of new tasks (initial deployment). In this situation, the client (the message-passing environment) requests from *amigod* instructions about where and how the new tasks should be placed among the distributed nodes. Such request is relayed to the proper policy, and the response is returned to the client, which does the deployment. The second situation happens when a preemptive policy, with process migration support, has the initiative of redistributing the load of the nodes of the distributed parallel environment. In this situation, the policy instructs the client (through *amigod*) to transfer processes between nodes in order to achieve the expected distribution.

4. The JUMP Migration Policy

The use of initial deployment and process migration scheduling policies together offers better load balancing than using only one of them, since process migration can balance the load of the distributed parallel environment in a more dynamic way, through preemption [Furquim 2006].

In this paper, it is presented a new preemptive scheduling policy, named JUMP, which unifies initial deployment and process migration into a single policy. The policy aims for a better load balancing for CPU-bound applications in heterogeneous clusters. The name JUMP is a recursive acronym that expands to "***J****ump* ***U****nified* ***M****igration* ***P****olicy*".

When answering an initial deployment request, the policy tries to make decisions that will also benefit future process migration requests. The same happens when the policy activates process migration: the decision about which nodes will become receivers of migrated processes tries to benefit future initial deployment requests.

In order to achieve this goal, the policy periodically distributes and collects load information from all nodes of the environment and uses a heuristic in order to predict load changes for each node until the next load information cycle. For each load information cycle, the policy observes the consequences of previously taken decisions through the average of the number of running processes and the processing load index.

By observing the changes in the processing load index relative to the number of running processes for each node, the policy calculates the approximate average weight that each process represents in each node. This allows the policy to detect the heterogeneity of the cluster during its execution, and thus, is more appropriate for heterogeneous environments.

JUMP is a distributed scheduling policy and each node of the cluster runs one instance of the policy code, which serves requests originated locally. The instances of the policy exchange load information among themselves periodically, so that each instance knows the load collected by all other instances.

4.1. Load Information Management

The load information module of the policy operates in two cycles, a collection cycle and an information cycle. The collection cycle happens with a very short period (*Tc* = 1s, in the default policy configuration), and is responsible for collecting and accumulating three system metrics. The information cycle happens with a longer period (*Ti* = 15s), and it is responsible for calculating load indices from the collected metrics and distribute load information to the rest of the environment.

The load indices produced by this module are the number of executing tasks, the processing load and the processor usage. The number of executing tasks represent the number of active processes managed by the message-passing environment, considering all processes with no regards to their states (running, sleeping and blocked). The processing load represents the dispute for processing time through the number of threads in the operating system execution queue, normalized by the computational power of the node. The processor usage represents the ratio of effective processing time of processors.

Each instance of the policy broadcasts its local load information to other instances through *amigod*. When load information from a remote instance is received, the local instance of the policy calculates the weight of the remote node through an exponential moving average of the ratio between the processing load and the number of executing tasks. The smoothing factor of the exponential moving average is adjustable in the configuration of the policy, being 0.1 its default value. In the situation where the number of running tasks is equal to zero, the policy keeps the node weight previously calculated.

4.2. Initial Deployment Component

The initial deployment component of the policy is activated in response to a process-scheduling request received from *amigod*. The policy receives the number of tasks that should be created and identifiers of the architecture and the operating system of the receiving nodes that should be selected for scheduling. The policy determines the best distribution of tasks that satisfies the requirements and returns how many tasks should be created in each node of the environment.

For each node of the environment, the policy keeps information about its real and expected loads. The real load of the node represents the last load value received from the remote instance of the policy. The expected load is initially set to the real load, but varies as the local instance of the policy makes scheduling decisions, allowing a more precise scheduling until the next collection of load information from remote instances.

Tasks are distributed among the selected nodes in order to balance their loads, transferring tasks preferentially to nodes with the lowest loads, reducing the variance among the loads of the nodes. In an unbalanced cluster, this makes the load of less-loaded nodes to approximate to the cluster average load.

Load distribution is performed based on the expected load and the calculated weight of each node. The receiver of each requested task is selected iteratively, picking the node with the least expected load and updating it by summing the calculated weight of that node.

4.3. Process Migration Component

The process migration component of the policy is responsible for detecting the load unbalance over the cluster during the parallel application execution and for initiating the migration of one or more processes in order to redistribute the load.

During process migration, only a single node may act as a sender. Although the policy is distributed, there is a centralized component that controls which node is allowed to act as a sender, so that no two nodes will send processes to other nodes at the same time. After the node in greater need for load distribution is selected to be the first one to try migration, the algorithm proceeds in the selected node. This mechanism can be improved in future versions by using a distributed commit protocol in order to remove the dependency in a centralized component and increase fault resilience.

The process migration component is activated periodically, every minute. When activated, the nodes that are potential senders are selected according with a number of conditions. Potential senders must have a processor usage index near or equal to 100%, processing load index greater than one, number of executing tasks greater than or equal to one and must be able to migrate processes. The most loaded node has priority when initiating process migration.

Some checking operations are performed as part of the activation policy: if the node really is able to participate in the migration, if there is at least one other compatible node in the environment for process exchange and if there are local tasks that can be migrated. If any of these conditions is not met, the local instance of policy resigns its status of sender.

After the local instance of the policy decides its node is a sender and has permission to initiate migration, the next step is to determine the number of processes that should be migrated and select them. The number of processes is determined by the difference of the load index of the sender to the cluster load average. Based on the weight index of the node, a number of tasks are selected for migration as necessary in order to decrease the node load to the cluster average load. The processes selected for migration are those with the biggest processing time and smallest time elapsed since its creation. In other words, those processes created a short time ago (more likely to have caused the load unbalance) and which consumed the most amount of CPU time (in order to avoid selecting idle processes, which are unlikely to affect the node load).

Finally, receivers for the processes selected for migration are determined. The same selection procedure of the initial deployment component is used, but now restricted to a specific set of potential receivers, that are those compatible for process exchange (has the same architecture and operating system as the sender).

Like the initial deployment component, the process migration component uses the node weight and expected load indices in order to determine the receivers of migrated processes and to keep a prediction of the load changes of the other nodes.

After having determined which processes will be migrated and the receivers for each process, the local instance of the policy instructs the message-passing environment to perform the migration.

4.4. Implementation

JUMP was implemented as part of the AMIGO scheduling environment, allowing it to be used with any message-passing environment that was adapted for its integration with AMIGO. Among the message-passing environments with support to process migration available, currently only DPVM [Iskra et al. 2000] has compatibility with AMIGO, being the one chosen as development and test environment for the policy.

Other message-passing environments modified for its use integrated to AMIGO can also use the JUMP policy. In this case, the process migration component is disabled when the environment does not provide such support. This is a consequence of the uniform API provided by AMIGO to scheduling policies. Although allowed, this situation is not optimal and the user will not benefit from the full potential of JUMP if the environment does not provide support for process migration. New message-passing environments may use the JUMP policy given they are adapted to work together with AMIGO.

5. Performance Evaluation

There are many different ways to perform process scheduling in distributed systems, using different tools and scheduling policies. Each policy may be more suitable for some specific situations than others. Performance evaluation provides secure statements with regards to the practical benefits of some tools and policies when compared to others, in specific situations.

One way to verify the end user satisfaction with a new configuration is through a smaller response time of the distributed application being executed. A smaller response time is obtained by optimizing the environment resource usage, which is a direct consequence of an effective load balancing. This is the goal of the JUMP scheduling policy. In this paper, the response time is considered as the elapsed time from when the user launches the distributed parallel application until when the application finishes its processing and presents the result back to the user.

In this section, the experiments performed in order to evaluate and analyze the performance of the JUMP scheduling policy are described.

5.1. Platform Description

All experiments were executed using a cluster of four personal computers with IBM/PC x86 32-bits architecture. The nodes have non-uniform configuration, with differences in processing power and amount of memory. Thus, it is a heterogeneous cluster and was chosen because it is the environment where the benefits of process migration are more evident.

The cluster is composed of a master node, Intel Pentium 4 3.00 GHz, 512 MB of memory and 40 GB of hard disk storage. The slave nodes vary in configuration, from an Intel Pentium II to an Intel Pentium 4, from 128 MB to 256 MB of RAM and no hard disks (using network boot).

All computers run the Linux 2.4.26 operating system kernel, with the *glibc 2.3.2-dynckpt* system library. This configuration is required by the message-passing environment DPVM 2.0. The cluster does not share users and was specially prepared so that each node runs only the most fundamental services and thus minimizing the influence of external processes on the evaluation.

The computers were interconnected by a FastEthernet network (100 Mbps), with a switch using the same technology and full-duplex communication.

5.2. Applications Used

Since the scheduling policy developed in this work aims at optimizing the performance of applications that demands intensive processing, three such applications were chosen for performance evaluation. Even among CPU-bound applications there are different execution profiles that may affect the quality of the process scheduling. Three applications with different execution profiles were chosen.

The first application used is a numerical calculus application which uses the composite trapezes method to calculate the result of a definite integral. The application demands intensive processing with floating point calculations and a very small amount of communication.

The second application calculates the product of two very large matrices. Like the previous application, this one demands intensive processing with floating point calculations, however, this application uses a lot more interprocess communication.

The third application performs Mandelbrot set fractal rendering in high resolution. This application has an execution characteristic that is very different from the other two. Each process of the application is entrusted to rendering a region of the fractal. The process estimates the amount of calculations that will be necessary to draw that region and, if a threshold is reached, the process creates new tasks in order to partition its work. Thus, the application forms a hierarchical structure of distributed processes.

5.3. Experiments Performed

Many sets of factors could be considered for performance evaluation. Since approaching all possible sets would result in a large multidimensional problem, the experiments performed tried to cover some of the combinations.

The factors considered in the work described in this paper were the scheduling policies used, the load level of the system and the applications used. These factors cover a large variety of possible uses of the JUMP policy and provide a good general view of its performance.

The performance of the JUMP policy is compared with other two policies implemented in AMIGO: round-robin and DPWP (Dynamic Policy Without Preemption) [Araújo et al. 1999]. The round-robin policy represents the trivial scheduling used by default by many message-passing environments, such as PVM [Geist et al. 1994], DPVM [Iskra et al. 2000] and LAM/MPI [Burns et al. 1994]. DPWP is a distributed scheduling policy for processing-intensive applications that uses the number of processes in the operating system execution queue as a load index, classifying each node according to its load in “idle”, “moderate” and “loaded”, distributing tasks to lesser loaded nodes. Both policies don’t support migration.

Three different system load situations were considered: idle system, loaded system and system with varying load. Each application was executed 30 times in each scenario, and the arithmetic average of the response times was calculated.

The warm-up technique was used during the experiments in order to allow the JUMP policy to detect and learn the heterogeneity of the cluster, providing better results. Therefore, an additional number of executions were performed before the 30 executions effectively measured in each of the scenarios.

The Hypothesis Test [Lehmann and Romano 2005] was used in order to verify the statistical significance of the results obtained during the policy performance evaluation. The significance level used in the tests was 0.05, and the test used was the comparison of the averages of two populations for independent samples.

Initially, the JUMP policy was compared with the performance of the other two policies in an idle system, that is, the only application running in the cluster was the application being evaluated itself.

The obtained results are shown in Figure 1. In this scenario, JUMP presents a significant performance gain compared to the other two policies. There is an observable variance in the response time of the applications for the JUMP policy, which is attributed to the synchronism of the applications with the process migration. Process migration is activated periodically with a frequency of once per minute. When process migration happens, the application can be near the beginning of its execution (thus being very benefited with the load balancing in its response time) or near the end (and virtually irrelevant or even damaging for its response time due to the migration overhead).

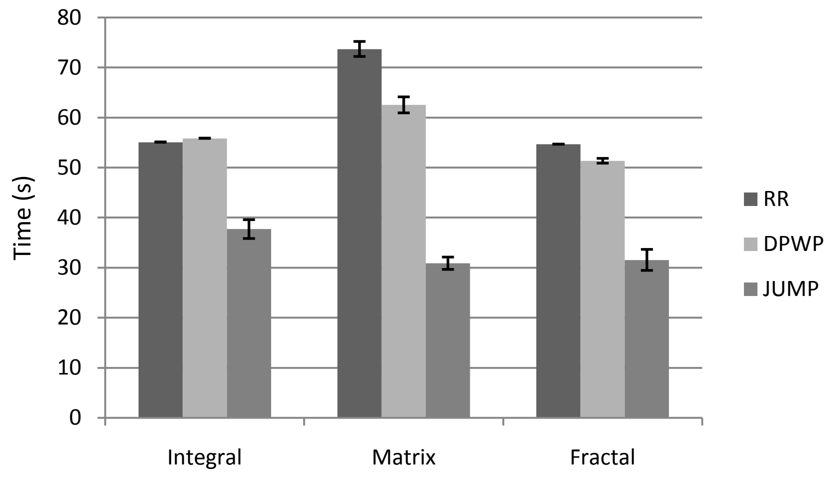

Figure 1.Comparing round-robin, DPWP and JUMP policies, idle system.

In the second scenario, JUMP was compared to the other two policies in a previously loaded system. The same tests performed in the previous experiment, with the same parameters for each application, were performed upon this new scenario. Before running each test of this experiment, each computer was loaded with three dead weight processes, not managed by the message-passing environment. Each process keeps constantly demanding processing time until the end of the test.

The results obtained in this second experiment are shown in Figure 2. Both policies DPWP and JUMP presented better performance than round-robin for all applications. The JUMP policy presents a better performance than DPWP for the integral and fractal applications. For the matrix application, the JUMP policy does not reflect the same gains observed for the other two applications, having an inferior performance than for the DPWP policy.

By investigating the behavior of the matrix application when scheduled by each of the policies, it was observed that a considerable amount of execution time for each process is spent with communication between nodes, when slave processes transmit their partitions of the resulting matrix to the master process. The JUMP policy schedule the processes in a way to keep the load of nodes well distributed throughout the cluster, and as a consequence, slave processes finish their execution with great synchronism, causing a peak in network utilization. The DPWP policy, on the other hand, schedule the processes in a less rigid way, making their communication more interleaved and not causing network usage peaks.

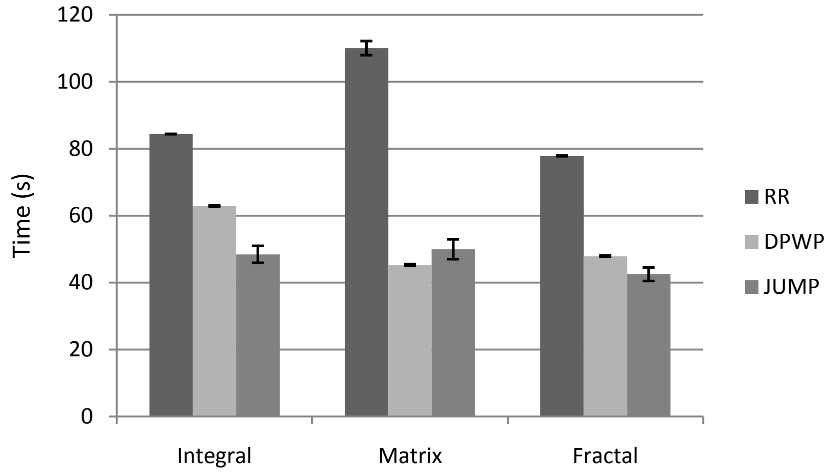

Figure 2.Comparing round-robin, DPWP and JUMP policies, loaded system.

Finally, the scheduling policies were compared in a system where the load varies during the execution of the parallel application. During the execution of each test of this experiment two computers of the platform were loaded with dead weight processes that were not managed by the message-passing environment.

The results obtained in this experiment, shown in Figure 3, evidence precisely the best situation where a scheduling policy with process migration support is most efficient.

The results provided by DPWP are similar to the ones from round-robin. Since the policy does not support preemption, decisions of initial deployment are based on the initial state of the cluster (idle), and continue until the end of the application. The JUMP policy, on the other hand, presents an excellent performance in all cases, since the policy is able to detect the load unbalance and relocate processes in order to equalize it.

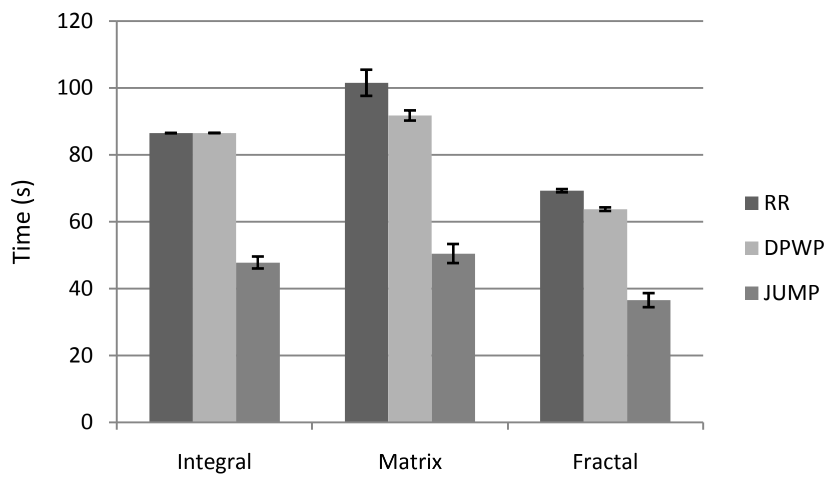

Figure 3.Comparing round-robin, DPWP and JUMP policies, system with varying load.

6. Conclusion and Future Work

This paper presented a new process scheduling policy for dynamic load balancing in distributed parallel environments with unified support for initial deployment and process migration. The policy is designed to be used for CPU-bound applications in heterogeneous clusters.

The results obtained through performance evaluation showed that the policy presents better application response time than the other policies implemented in the scheduling environment used, for most of the observed scenarios, with multiple system load situations and multiple applications.

The results presented in this paper are expected to incite research on other new policies with unified initial deployment and process migration mechanisms. For example, more generalized policies that also consider memory usage and communication in addition to processing load. Other goals to process migration may also be explored, like fault tolerance and system administration.

Other future work includes porting other process migration policies and environments to the same scheduling environment used, enabling the evaluation of JUMP against other process migration policies, and performance indices specific to such policies.

Acknowledgments

The authors acknowledge the National Council for Scientific and Technological Development (CNPq), for the financial support that allowed the production of this work, as well to FAPESP and to CAPES for the support provided to works developed by the Distributed Systems and Concurrent Programming Group from ICMC-USP.

References

Araújo, A. P. F., Santana, M. J., Santana, R. H. C. and Souza, P. S. L. (1999) "DPWP – a new load balancing algorithm". 5th Intl. Conf. Information Systems Analysis and Synthesis – ISAS'99, Orlando, U.S.A.

Burns, G., Daoud, R. and Vaigl, J. (1994) “LAM: an open cluster environment for MPI”, Proc. Supercomputing Symposium. pp. 379-386.

Campos Jr., A. (2001) “Development of a graphical interface for a scheduling environment” (orig.: “*Desenvolvimento de uma interface gráfica para um ambiente de escalonamento*”). University of São Paulo, São Carlos.

Carzaniga, A., Picco, G. P., and Vigna, G. (1997) “Designing distributed applications with mobile code paradigms”. Proc. 19th Intl. Conf. Software Engineering – ICSE'97, pp. 22-31, ACM Presss.

Douglis, F. and Ousterhout J. (1991) “Transparent process migration: design alternatives and the Sprite implementation”, Software - Practice and Experience, vol. 21, no. 8, pp. 757-785.

Figueiredo, T. C.; Santana, M. J.; Santana, R. H. C.; Souza, P. S. L. (2002) “Improvements on the Performance of LAM/MPI Applications: The use of the Amigo Environment for Efficient Process Scheduling” (orig.: “*Melhorias no Desempenho de Aplicações LAM/MPI: Uso do Ambiente Amigo para o Escalonamento Eficiente de Processos*”). WPerformance, Florianópolis, 2002. v. 1. pp. 1358-1368.

Fuggetta, A., Picco, G. P. and Vigna G. (1998) “Understanding code mobility”. IEEE Trans. Software Engineering, vol. 24, no. 5, pp. 342-361.

Furquim, G. A. (2006) “Load balancing of SPMD applications in distributed computational systems” (orig.: “*Balanceamento de cargas de aplicações SPMD em sistemas computacionais distribuídos*”). University of São Paulo, São Carlos. Available at: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-18072006-173002/>

Geist, A., Beguelin, A., Dongarra, J. J., Jiang, W., Manchek, R. and Sunderam, V. (1994) “PVM: parallel virtual machine. A Users’ Guide and Tutorial for Networked Parallel Computing.”. MIT Press, Cambridge, Massachusetts.

Gray, R. S., Cybenko, G., Kotz, D., Peterson, R. A. and Rus, D. (2002) “D'Agents: applications and performance of a mobile-agent system”. Software - Practice and Experience, vol. 32, no. 6, pp. 543-573.

Iskra, K. A., Hendrikse, Z. W., Van Albada, G. D., Overeinder, B. J. and Sloot, P. M. A. (2000) “Dynamic migration of PVM tasks”. Proc. Sixth Annual Conf. of the Advanced School for Computing and Imaging, pp. 206-212.

Lehmann E. L. and Romano J. P. (2005) “Testing statistical hypotheses”. ISBN 0387988645.

Milojičić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S. (2000) “Process migration”, ACM Computing Surveys, vol. 32, no. 3, pp. 241-299.

Noguchi, K., Dillencourt, M. B. and Bic, L. F. (2008) “Efficient global pointers with spontaneous process migration”. 16th Euromicro Conf. Parallel, Distributed and Network-Based Processing – PDP 2008, pp. 87-94.

Richmond, M. and Hitchens, M. (1997) “A new process migration algorithm”, ACM SIGOPS Operating Systems Review, vol. 31, pp. 31-42.

Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove P. and Roman E. (2005) "The LAM/MPI checkpoint/restart framework: system-initiated checkpointing". Intl. J. of High Performance Computing Applications, vol. 19, no. 4, pp. 479-493.

Santos, R. R.; Santana, M. J. (2001) "Parallel application scheduling: AMIGO-CORBA interface" (orig.: "*Escalonamento de aplicações paralelas: interface AMIGO-CORBA*"). Workshop de Teses e Dissertações Defendidas, 2001. University of São Paulo, São Carlos.

Shivaratri, N. G., Krueger, P. and Singhal, M. (1992) "Load distributing for locally distributed systems". IEEE Computer, vol. 25, no. 12, pp. 33-44.

Souza, P. S. L., Santana, M. J. and Santana, R. H. C. (1999) "AMIGO – a dynamical flexible scheduling environment", 5th International Conf. Information Systems Analysis and Synthesis – ISAS'99, Orlando.

Voorsluys, B. L. (2006) "Influences of scheduling policies on the performance of distributed simulations" (orig.: "*Influências de políticas de escalonamento no desempenho de simulações distribuídas*"). University of São Paulo, São Carlos. Available at: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-13092006-162607/>

Wang, C., Mueller, F., Engelmann, C. and Scott, S. L. (2008) "Proactive process-level live migration in HPC environments". Proc. 2008 ACM/IEEE Conf. on Supercomputing.