“Virtual Malleability” applied to MPI jobs to improve their execution in a multiprogrammed environment

Gladys Utrera Iglesias

Departament d’Arquitectura de Computadors

Universitat Politècnica de Catalunya (UPC)

Barcelona (SPAIN)

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la Universitat Politècnica de Catalunya

“Virtual Malleability” applied to MPI jobs to improve their execution in a multiprogrammed environment

Author: Gladys Utrera Iglesias

Co-Advisor: Julita Corbalán González

Advisor: Jesús Labarta Mancho

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: ....
Autor de la tesi: ....

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, …………… de/d’…...............……………. de ....…

El President	El Secretari	
.... (nom i cognoms)	.... (nom i cognoms)	
El vocal	El vocal	El vocal
.... (nom i cognoms)	.... (nom i cognoms)	.... (nom i cognoms)

Al fin y al cabo somos lo que hacemos para cambiar lo que somos.

Eduardo Galeano

Abstract

This work focuses on scheduling of MPI jobs when executing in shared-memory multiprocessors (SMPs).

The objective was to obtain the best performance in response time in multiprogrammed multiprocessors systems using batch systems, assuming all the jobs have the same priority.

To achieve that purpose, the benefits of supporting malleability on MPI jobs to reduce fragmentation and consequently improve the performance of the system were studied.

The contributions made in this work can be summarized as follows:

- *Virtual malleability: A mechanism where a job is assigned a dynamic processor partition, where the number of processes is greater than the number of processors. The partition size is modified at runtime, according to external requirements such as the load of the system, by varying the multiprogramming level, making the job contend for resources with itself.*

  In addition to this, a mechanism which decides at runtime if applying local or global process queues to an application depending on the load balancing between processes of it.

- *A job scheduling policy, that takes decisions such as how many processes to start with and the maximum multiprogramming degree based on the type and number of applications running and queued. Moreover, as soon as a job finishes execution and where there are queued jobs, this algorithm analyzes whether it is better to start execution of another job immediately or just wait until there are more resources available.*

- *A new alternative to backfilling strategies for the problema of window execution time expiring. Virtual malleability is applied to the backfilled job, reducing its partition size but without aborting or suspending it as in traditional backfilling.*

The evaluation of this thesis has been done using a practical approach. All the proposals were implemented, modifying the three scheduling levels: queuing system, processor scheduler and runtime library.

The impact of the contributions were studied under several types of workloads, varying machine utilization, communication and, balance degree of the applications, multiprogramming level, and job size.

Results showed that it is possible to offer malleability over MPI jobs.

An application obtained better performance when contending for the resources with itself than with other applications, especially in workloads with high machine utilization. Load imbalance was taken into account obtaining better performance if applying the right queue type to each application independently.

The job scheduling policy proposed exploited virtual malleability by choosing at the beginning of execution some parameters like the number of processes and maximum multiprogramming level. It performed well under bursty workloads with low to medium machine utilizations.

However as the load increases, virtual malleability was not enough. That is because, when the machine is heavily loaded, the jobs, once shrunk are not able to expand, so they must be executed all the time with a partition smaller than the job size, thus degrading performance. Thus, at this point the job scheduling policy concentrated just in moldability.

Fragmentation was alleviated also by applying backfilling techniques to the job scheduling algorithm. Virtual malleability showed to be an interesting improvement in the window expiring problem. Backfilled jobs even on a smaller partition, can continue execution reducing memory swapping generated by aborts/suspensions In this way the queueing system is prevented from reinserting the backfilled job in the queue and re-executing it in the future.

A Nieves, Elvio, Jesusa y Leo

Acknowledgements / Agradecimientos

I want to thank all the people who provided guidance, help and support and make this work possible.

To my advisors Julita Corbalán and Jesus Labarta, for the their guidance during all this period, the valuable comments and advices, for their support and for letting me freedom to do things I wanted to do in the way I believed they should be done.

I have to thank the members of my thesis committee for the time and effort put on judging this work.

I would like to express my gratitude to Mark Bull for having given me the chance to work with him and for his suggestions and ideas which helped to improve this document. I have to thank all the staff of the EPCC at the University of Edinburgh for their kindness hosting me there, and in this particular I have to mention specially Catherine Inglis. And of course to Mario Antonioletti for his suggestions and English corrections, and to Adam Carter.

I am also especially grateful to the people of the Computer Architecture Department. Thanks to Eduard Ayguadé and Mateo Valero for their encouragement in many different situations. I want to thank also to Xavier Martorell for his advices and help during these years.

I also would like to express my gratitude to the people of administration, especially to Trini for her patience. And I need to thank the staff of the LCAC and CEPBA for their excellent management and technical support.

Finally, I would like to thank the staff at the InCo, at the Faculty of Engineering, in Montevideo who provided the initial support that allowed me to start my Phd at Barcelona.

I need to thank everyone I have not cited above but have helped me in a direct or indirect way during all this period.

Y todo éste trabajo no hubiera sido posible sin el apoyo y soporte emocional de las personas que me han acompañado durante todo este tiempo, ya sea cerquita o a la distancia, por lo tanto es mérito de ellas también.

En primer lugar a mis padres Nieves y Elvio, que son lo más lindo que tengo, no hay palabras para expresarlo. A la abuela Jesusa, que desgraciadamente no me pudo acompañar hasta aquí … a mi hermano Leo, mis primitos de acá, Isabel mi mejor amiga, mis primitos de allá, tías, tíos, a mi familia en general. A todos ellos, muchas gracias por el amor, cariño, apoyo incondicional y confianza.

A los amigos y compañeros del departamento que he tenido a mi lado durante todo éste tiempo compartiendo innumerables charlas, discusiones y cafés. En especial a Germán Rodríguez, Rubén González, Miquel Pericàs, Manel Fernández, Pau Artigas y Montse Farreras.

A los de acá: Josep Carmona, Guillem Godoy; a los de allá: Eduardo Grampín que motivó ésta aventura, Marcelo Bertalmío, Serrana Cabrera, Javier Baliosián, Italo Bove, Rodrigo Calvete y Vicky Alcoba.

A todos gracias por estar ahí.

This work was supported by the CIRI in the "Donación para el CEPBA-IBM Instituto de Tecnología" project, the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01 and TIN2007-60625, and the HiPEAC European Network of Excellence, And has been developed using the resources of the DAC at the UPC and the European Centre for Parallelism of Barcelona (CEPBA).

Index

Chapter 1

INTRODUCTION

Abstract

This section describes the motivation of this work. In addition there is a description of each of the proposals made. At the end there is a quick look on the execution environment built to implement, evaluate and compare all the contributions of this work.

1.1 Introduction

An operating system must give support to different kind of applications, such as parallel, sequential, I/O intensive and batch. The scheduler has to take into account the particular characteristics of each architecture and each job, to exploit the maximum performance of the overall system. Shared-memory multiprocessors (SMPs) are the backbone of SMP clusters. Simultaneous multithreading (SMT) and multi-core/chip-multiprocessing (CMP) SMPs are the emerging architectures.

This work focuses on improving the scheduling of parallel jobs when executing in shared-memory multiprocessors. A parallel job is an application composed of processes which run concurrently and cooperate to do a certain computation. Parallel jobs are characterized by having processes that communicate and synchronize with each other.

To achieve the purpose of this work, it was necessary to have a batch queuing system which was in charge of dispatching the jobs that arrive, a processor scheduler which managed the processor partition and mapping, and a runtime library which managed the processor sharing among processes from the jobs.

Depending on the ability of parallel jobs to adapt to changes in resource availability changes and allocated resources, they can be classified [FRSS97] as: *rigid, moldable* and *malleable*. A job is said to be rigid when its number of processes is specified external to the scheduler and it must remain fixed during execution. A job is said to be moldable when the decision over the number of processes can be delayed until the beginning of the execution. However, once it starts executing, this number cannot be modified. A job is said to be malleable if there exists also the possibility to modify the number of processes during execution.

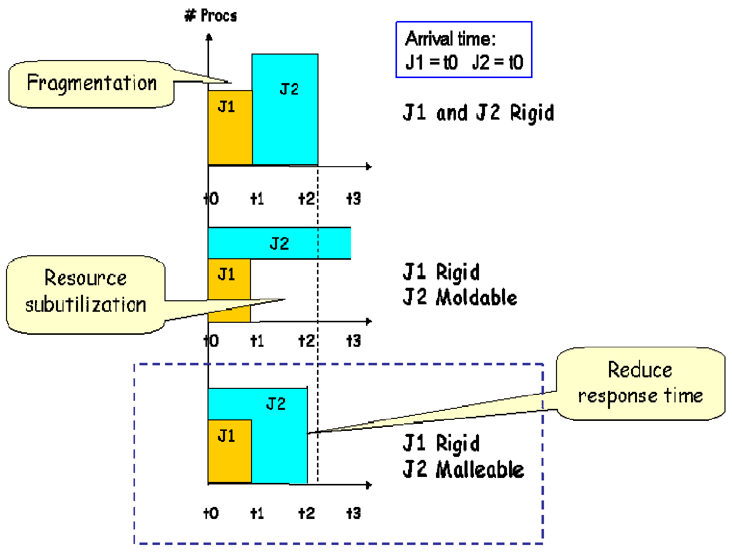


Figure 1.1 **Comparing execution of rigid, moldable and malleable jobs**

Figure 1.1 shows three examples of execution of two jobs when they are both rigid, when one of them is moldable and finally when one of them is malleable. As can be seen rigid jobs generate fragmentation which is completely alleviated when there are malleable jobs. They are able to adapt to changes in the system when other jobs finish and free resources. In addition, response time is reduced.

The objective of this work was to obtain the best performance in response time and throughput in multiprogrammed multiprocessors systems when working with parallel jobs running in batch systems, assuming all the jobs have the same priority.

1.2 The Problem

In order to get better machine utilization and synchronization among processes from a parallel job, a typical scheduling strategy in parallel systems is to allocate jobs into processor partitions for their exclusive use; these are *space-sharing* policies [GuTU91].

If parallel jobs are rigid, space-sharing policies allocate static processor partitions for them. These can suffer from fragmentation [WeFe01], which can be alleviated by applying backfilling strategies [ShFe03], which consist of bringing forward short jobs in order to take advantage of free processors provided that they will not delay previously queued jobs.

Moldability [Cirn01] can also reduce fragmentation because jobs are sized to the available resources at the beginning of execution. However, this new facility has some drawbacks, for example they will remain fixed even though the system load varies, and not all the applications support any number of processes. In addition this facility is not commonly available in production systems.

Malleable jobs are the only ones that are able to adapt to load changes, eliminating the fragmentation completely. Applying dynamic scheduling policies helps to improve the system response time when the load varies.

The two most popular programming models for parallel jobs used in high performance computing centers are MPI [MPI94] and OpenMP [Open05]. Malleability has been demonstrated to work well over OpenMP [CoML00] jobs but it is not supported by MPI jobs. The MPI programming model is used worldwide, even on SMPs, due to its portability, compared to OpenMP.

In the case of OpenMP, malleability can be automatically offered by the runtime library because data re-distribution and remote data access is done transparently by the underlying hardware cache coherence mechanisms.

The case of MPI is much more complex because in MPI jobs, data is explicitly distributed across processes. Each time the number of processes is modified during execution, an explicit data re-distribution has to be programmed. This data re-distribution requires a deep knowledge of each particular application by the MPI runtime library.

1.3 Our work

This dissertation studies the benefits of supporting malleability on MPI jobs to reduce fragmentation and consequently improve the performance of the system.

As is already known, dynamic processor partitioning, has been demonstrated to work well for parallel job scheduling. This work aims to show that it is possible to offer Malleability over MPI jobs. To achieve that purpose several mechanisms were developed which involve the job scheduling level, the processor scheduling level and a runtime library which manages the processor sharing.

The contributions made in this work can be summarized as follows:

- *Virtual malleability*: A mechanism for efficient execution of MPI jobs, where a job is assigned a dynamic processor partition, when the number of processes is greater than the number of processors. The partition size is modified at runtime, according to external requirements such as the load of the system, by varying the multiprogramming level, making the job contend for resources with itself [UtCL0904]. In addition to this a mechanism was also developed to make a runtime decision about applying a global process queue per application or local process queues per processor, taking into account the application load balancing degree. This mechanism measures the load balancing degree of each application, and after that applies immediately the relevant queue type. As shown below in Figure 1.2, as soon as job B finishes execution and free its processor partition, job A is able to expand and take advantage of the newly available processors [UtCL0905].

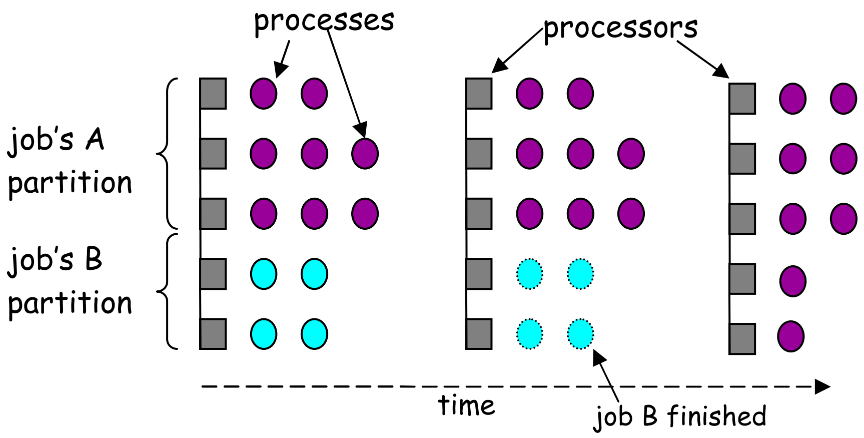


Figure 1.2 **Processor allocation applying virtual malleability**

- *Folding by job type* (FJT) [UtCL1004]: A job scheduling policy which takes decisions concerned with the number of processes of an application and the maximum multiprogramming degree based on the type and number of applications running and queued. Moreover, as soon as a job finishes execution and where there are queued jobs, this algorithm analyzes whether it is better to start execution of another job immediately or just wait until there are more resources available.

- The addition of backfilling to FJT, and the application of *virtual malleability* to backfilling policies [UtCL0605], resulting in an improvement of response time of the overall system.

1.4 Contributions of this work

As already mentioned, virtual malleability, is a strategy which enables MPI jobs to execute in dynamic processor partitions. In addition, to take the maximum advantage of this facility in system throughput and response time, a job scheduling algorithm to be applied at the queuing system level was designed. That is, once a job arrives in the system, this algorithm, taking into account information such as machine load, queued jobs, and expected execution time, decides when to execute the job, the initial number of processes, and the multiprogramming level. Virtual malleability was also applied to existing job scheduling policies such as the *backfilling* [Lift95] to improve its performance [AnLL89].

In this section the contributions of this work summarized in the previous section, as well as the mechanisms involved are described in detail.

1.4.1 Virtual Malleability

Virtual malleability arises from the combination of moldability in order to decide the number of processes, and a mechanism proposed in this work, *Self coscheduling* [UtCL0904], to make the job's partition modifiable at runtime.

1.4.1.1 Self coscheduling (SCS)

The self coscheduling [UtCL0904] is a mechanism that exploits the low-level process management with the goal of minimising the loss of performance generated when the number of total processes in the system is greater than the number of processors. This is the case when the multiprogramming level (MPL) is incremented in order to increase machine utilization.

It was demonstrated that it was possible to combine *coscheduling* policies [ArCu01],[Feit94],[NBSD99] with *space-sharing* policies [GuTU91] to build a dynamic space-sharing scheduling policy, which was named in this work as *self coscheduling* policy (SCS) [UtCL0904], in a dynamic environment where the number of processors allocated to a job may be modified during job execution.

Coscheduling approaches are based on scheduling the largest possible number of communicating processes from a job simultaneously, in order to overcome the synchronization problem. Coscheduling techniques result from the combination of components in the interaction between scheduler and communication, which are related to what to do on message arrival, and how to wait for a message.

SCS was implemented, evaluated and compared to other implementations of coscheduling policies from the literature such as *periodic boost* [NBSD99], *spin blocking* [ArCu01] and pure time-sharing.

It was observed that the execution of a job had better performance when competing for the resources with itself than with other jobs.

1.4.1.2 Runtime decision about local & global process queues

As the number of processes could be greater than the number of processors, it was studied how to organize processes; that is whether to choose the next process to run from a local queue per processor, or from a global process queue per application.

It is well known that the performance of one or the other approach depends on the load balancing degree of the job [FeNi95]. A mechanism named *load balancing detector* (*LBD*) [UtCL0905] classifies at runtime an application and apply to it the approppiate queue type. In order to do that, the *LBD* measures the load balancing degree of each arrived job at runtime and then decide to apply a global queue in the case where the job has an imbalanced behavior, or local queues in the case where the job is well-balanced.

The beginning of the execution of any application is "chaotic" as processes are created and data is distributed. But, as soon as the execution becomes regular the mechanism calculates the coeficient of variation of the number of context switches per process, on a process global queue basis. This number is compared to an empiric pre-established border value to decide whether the the application has an imbalanced or well-balanced behaviour.

The load balancing behavior of several types of individual applications, and for different multiprogramming levels were studied. The mechanism was evaluated on those applications, and it was observed that for jobs with high communication degree and with mostly point-to-point communication, it performed acceptably. However for jobs with low communication degree or many collective communications it has no advantage.

Concerning the synchronization problem, after experimenting with several spin times, it was observed that blocking immediately obtained the best perfomance, which is spin time equal to zero. This is consistent with the fact that the experiments were run on a shared-memory machine, where the latency for message delivering may be considered as null [ArCu01]. When deciding the next process to run several heuristics were examined, such as the process which has the greatest number of unconsumed messages, round robin, etc. For local queues, the best option was to choose the next to run in a round robin fashion. For the case of global queues it was study also the process that has last run on that processor and the sender process of the message that hasn't arrived. The best performance was obtained by the heuristic related to the number of unconsumed messages.

1.4.2 Folding by Job Type (FJT)

Virtual malleability enables the job to adapt to the current conditions of the system. However, to take the maximum advantage of this facility it is necessary to specify some parameters at the beginning of the execution of the job. Such decisions must be taken by the queuing system in order to take the current system conditions into account.

The algorithm developed decides for each job at the beginning of execution the number of processes and the multiprogramming level, with the aim of reducing queuing time and improving system utilization. Another decision happens when the available processor partition is smaller than the job size. This means it must be analyzed to see if it will be advantageous to shrink some running jobs, including the one about to start, or just delay the start of execution (thus incrementing queuing time).

The extra information the algorithm need, is the classification of the job as either *long* or *short*. The user has to estimate this before launching the job. This type of classification is commonly used in production systems where a job is submitted by a user to a queue depending on its required number of processors, its estimated execution time and other parameters. In spite of being a quite simple classification, with just two categories, they were considered enough for the study since the objective was just to measure the impact of varying partition sizes when jobs from different execution times arrive in the system. In addition to this, as the experiments were based on real executions with exclusive use on the processors, there were practical limitations on the duration of each workload, they couldn't last for days, at most one or two hours. However, in the literature, the classification of jobs varies from 2 to 4 classes; those are long and short, or long, very long, short and very short.

Long jobs must be run with the maximum possible number of processes. Short ones execute during very short time, so they must not prevent a long one starting execution with the largest possible number of processes. This is a typical situation where it was applied virtual malleability in order to adapt the partition size of the long job according to the current situation. The algorithm was named *folding by job type* (*FJT*) [UtCL1004].

In order to take these decisions, the algorithm examines whether the new job is long or short, whether there are long and/or short jobs running, if there are long jobs queued, and/or short jobs queued and how many in each case.

The idea is that each time a job arrives in the system, *FJT* deduces available information from the current context, such as the class of the queued and running jobs, whether it is possible that in the near future, more processors will become available.

This can happen if one of the situations occurs: 1) there are short jobs running and the MPL=1, this means that currently there aren't any jobs running shrunk; 2) there aren't any long jobs queued. When one of these two conditions holds, and a long job arrives, *FJT* will assign to it a number of processes greater than the number of available processors. They will run shrunk until short jobs finish execution. After that, the long job will be able to expand to the newly freed processors. Notice that queuing time is reduced and the long job is able also to take advantage of the resources freed later. On the contrary, if a short job arrives and there are no idle processors, then if there are long jobs executing they can be shrunk in order to free processors and let the short job start execution immediately.

FJT was implemented and compared to an implementation of *folding* combined with moldability [PaDo96] and some pure moldability techniques [Cirn01][RSS99], such as *ASP* and *PSA*. All of the policies were evaluated under workloads with different job sizes, classes and machine utilization.

Results showed that *FJT* adapts easily to load changes. The proposal has benefits especially when load varies strongly. As the jobs start execution shrunk (i.e. with a partition smaller than the number of processes), then if the load goes down there are available processors, and the job is able to expand to the newly freed processors. This situation is very common in workloads with arrival bursts.

1.4.3 Folding by Job Type with backfilling

Virtual malleability reduces fragmentation by adjusting the partition sizes according to the available resources during the execution of the jobs. But, as the load incremented the system performance degraded significantly because jobs were not able to expand and had to run shrunk during the whole execution. In such a situation, moldability was enough. However, there are jobs that are not "fully moldable" as they can only run on certain number of processors (for example perfect squares, powers of two). In addition there must be a compromise between reducing wait queue time and incrementing execution time. For example a long job degrades execution time when executed on a small processor partition.

For these reasons, when working with such heavy load systems, in spite of applying moldability fragmentation is not eliminated at all. Backfilling [Lifk95] techniques can alleviate the fragmentation by filling holes generated in the situations described above.

As backfilling techniques rely on user runtime estimates, there may be inaccuracy. If a backfilled job doesn't finish execution within the window time assigned, it will prevent the job at the head of the queue from executing. To treat this problem, the backfilled job can be: 1) aborted [SMCJ02], 2) suspended/resumed, 3) checkpointed/restarted [SMCJ02], 4) remain executing during a period of time [TaFe99], [WaMW02]. Except for option 4), the scheduler will have to reinsert the job in the wait queue. In option 2) the job must be resumed in the same processor partition, unless it is running on a shared memory multiprocessor, in which case it is still advisable to minimize the memory impact. This may add a considerable delay for resuming the job. In addition not all operating systems have support for option 3).

By applying the concept of virtual malleability, this work proposes a new alternative to the window expired job problem [UtCL0905]. The execution is not aborted, nor is the execution of the job at the head of the queue delayed. The partition size of the backfilled job is just reduced by applying virtual malleability to it, shrinking the processor partition of the job. In this way, the backfilled job is made "malleable", thus freeing resources for the highest priority job. It is important to notice that if the backfilled job had aborted there would be even more free resources. In the proposed scheme, as jobs can be moldable, such differences are adjusted to the newly available partition.

As a result the job don't have to be reinserted in the queue and the backfilled job is able to continue execution, minimizing delays because of inaccuracy of runtime estimates.

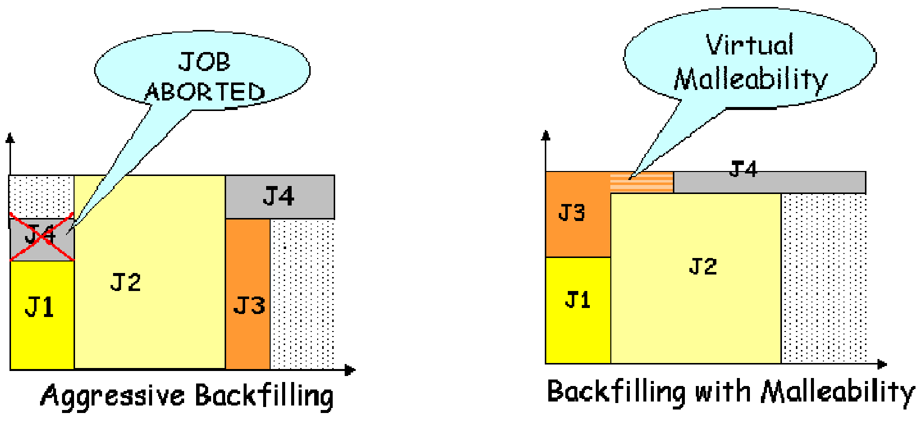


Figure 1.5 Moldable jobs with traditional backfilling (left) and backfilling with malleability (right)

The proposal of this section, backfilling with malleability, was implemented and compared with other moldability and backfilling techniques, under several dynamic workloads and demonstrated a performance improvement of about 20 to 30% especially for high machine utilization.

The strategy is portable and can be supported by any operating system. It reduces memory swapping generated by aborts/suspensions, prevents the queuing system from reinserting in the queue and re-executing the job in the future. It has to be noticed that if the job is reinserted in the queue it becomes eligible to be backfilled again.

1.5 Overview of the execution framework

The evaluation of this work has been done using a practical approach. The proposals were implemented, modifying the three scheduling levels: queuing system, processor scheduler and runtime library.

The impact of the contributions of this work was studied by evaluating them under several types of workloads, varying machine load, communication degree of applications, balance degree of applications, multiprogramming level, and job size. Results showed that the ideas proposed in this work of applying virtual malleability to MPI jobs obtained an acceptable performance in most cases.

1.5.1 Execution environment

This section describes the software architecture which is in charge of implementing the scheduling policies described in this work.

The execution environment is composed by a resource manager or *CPU Manager* (*CPUM*), a queuing system and a runtime library (VMruntime). Each time a job arrives in system, the queuing system takes control of it and decides the maximum multiprogramming level, the number of processes, the order in the queue and if it must be run immediately or delay execution. The queuing system coordinates with the *CPUM*. The

CPUM is a resource manager which manages the processor allocation. The queuing system, named in this work *launcher*, and the *CPUM* communicates via shared memory. There is a runtime library, named VMruntime, which is in charge of the process mapping and scheduling. In order to do that it wrappers the MPI calls to the library and a system call (*sginap*). This is useful for tracking information about number of messages arrived, sent, processes blocked on a message, number of context switches.

1.5.2 Queuing system: Launcher

The *launcher* is the user-level queuing system used in our execution environment. It performs the scheduling dispatching policy from a list of jobs belonging to a predefined workload, which is received as a parameter.

The launcher is informed about the job class (long or short) as well as the range of possible initial number of processes. The *CPUM* knows about the resource availability.

Once the launcher has chosen the job to launch from the wait queue it decides the optimal number of processes and the multiprogramming level according to the processor allocation policy. The launcher knows the job class of all the jobs in the system: those from the wait queue and running jobs.

1.5.3 Cpu Manager (CPUM)

The *CPUM* is a user-level scheduler. The launcher indicates to the *CPUM* the number of processes and the MPL allowed for each job. Taking into account all this information, the *CPUM* implements the processor allocation policies, deciding where the job will be allocated and its processor partition size.

Once the queuing system launches a job, it starts execution if there are enough free processors that satisfy the minimum requirement calculated by the *CPUM*. Otherwise it must wait until other job finishes execution and free processors were recalculated. During execution, if there aren't any queued jobs, then free processors are redistributed, among the jobs in the system. As soon as a new job arrives if there aren't enough free processors for it, all the jobs are shrunk again.

The *CPUM* wakes up periodically, and at each quantum expiration examines whether new jobs have started or finished execution and updates the control structures and do the processor allocation.

1.5.4 Application Runtime Library (VMruntime)

The runtime library, in order to get control of MPI jobs, uses the *ditools library* [SeNC00]. It consists of a dynamic interposition mechanism that intercepts functions such as the MPI calls or a system call routine such as *sginap*, which is invoked by the MPI library when it is performing a blocking function. These functions provide information to the VMruntime, or get information from it. In addition, using this mechanism, the execution of the sginap routine is inhibited. This is useful when having several processes allocated to a processor, as when applying virtual malleability. The sginap wrapper is in charge of doing context switching each time the spin time has expired and decides which process runs next. It is important to notice that if the spin time is equal to zero blocking immediately will be the case. The interposition mechanism is also used to initialize some control structures of the application runtime library and to find out each process MPI rank.

The VMruntime is also in charge of the process to processor mapping. It decides at runtime to apply local processes queues per processor or global process queues per application, depending on the load balancing degree of it. All the techniques were implemented without modifying the source code of the MPI library and without recompilation of the applications.

1.6 Organization of the work document

The rest of the chapters are organizad as follows:

Chapter 2 describes the main elements of the architecture that forms part of the execution environment of this work. Those are the multiprocessor system, scheduling policies, multiprogramming models.

Chapter 3 gives a detailed description of the execution framework of this work. It describes the implementation and functioning of the queuing system, the resource manager and the runtime library. There is also an overview of related work about those topics.

Chapter 4 presents the first proposal of this work, the virtual malleability mechanism. This mechanism that allows applications adapt to the availability of the external resources.

Chapter 5 continues with another proposal of this work, by scheduling policies at job level, the *FJT*. This is an algorithm which takes decisions related with the execution of a job from the system point of view.

Chapter 6 enhances the proposal of the last chapter by adding backfilling techniques and applying the concept of virtual malleability of chapter 4 to expired windows.

And finally, chapter 7 shows the conclusions and future work of this work.

Chapter 2

BACKGROUND

Abstract

The main components of a multiprocessor system are described in this chapter like architecture, operating system and programming models. It is analyzed the SGI 2000 which is the platform where the work was developed on. About the operating system, scheduling policies from the bibliography at job and processor level are described and process mapping schemes are also studied. There is discussion on programming models, including the one which is used in this dissertation: MPI.

Finally a job classification based on their flexibility to vary the size of the processor partition is described.

2.1 Introduction

This chapter gives an overview of the main components of a multiprocessor system: the architecture, operating system and the programming model, considering some commercial and open source implementations of these.

Some general characteristics of multiprocessor architectures are described, taking the SGI Origin 2000 as an example, as this was the actual machine on which the material presented in this work was implemented. This machine is a shared-memory multiprocessor (SMP).

The operating system is the other component that is considered in this chapter. In particular, the scheduling policies used form the main subject of this investigation. The contributions of this work lie mainly at job and processor scheduling level. Prior work in this area is also described.

Finally, some standard programming models are described including the one used in this work, the Message Passing Interface (*MPI*) [mpi]. Terminology that is applied to classify jobs depending on the flexibility they have to varying their number of processes they use at runtime and thus impacting on the scheduling policies that can be used, is defined at the end of this chapter.

This chapter is thus organized as follows: section 2.2 describes the main characteristics of multiprocessor architectures; section 2.3 presents an overview of traditional scheduling policies. Section 2.4 and 2.5 describes several programming models and a job classification scheme. Finally, section 2.6 summarises the main findings of this chapter.

2.2 Multiprocessor architectures

Multiprocessor architectures are important and widely used, with systems often found in supercomputing centres. These kinds of machines are characterized by having more than one processor. Even more, due to the existence of *VLSI* technology it is possible nowadays to find more than one processor on a single chip.

From the memory access point of view, it is possible to further classify multiprocessor architectures according to whether they use a unique global memory address space, that is shared-memory multiprocessors and those that use distributed shared-memory address spaces.

2.2.1 Shared-memory multiprocessor architectures

A shared-memory multiprocessor is a system that has several processors that share a unique global address space.

Although any processor can access the whole memory space, there is an additional characteristic: in some multiprocessors the memory can be accessed from any processor at the same cost regardless of how physically far it is from it. These are the UMA architectures

(*Uniform Memory Access*). The contrary of these are the NUMA architectures (*Nonuniform Memory Access*).

Processors and memory modules are connected through a bus. There are caches big enough help to minimize the network traffic as the data can be stored locally.

SMP is one of many styles of multiprocessor machine achitecture; others include NUMA (Non-Uniform Memory Access) which dedicates different memory banks to different processors allowing them to access memory in parallel. This can dramatically improve memory throughput as long as the data is localized to specific processes (and thus processors). On the downside, NUMA makes the cost of moving data from one processor to another, as in workload balancing, more expensive. The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users.

2.2.2 Distributed shared-memory multiprocessor architectures

In this type of architecture the programs have the illusion of addressing a unique memory space. This is achieved by applying a technique named *Distributed Shared Memory* (DSM) proposed in [LiHu89].

In DSM, in each time a processor tries to access a memory page that it is not locally available; it generates a call to the operating system. This locates the memory page and sends it across the network.

2.2.3 *CC-NUMA architecture*: *SGI Origin 2000*

The implementation and evaluation of this work was done on a distributed shared-memory with cache coherence (CC-NUMA), the SGI Origin 2000 [sgi00]. It has 64 processors organized in 32 nodes with two 250MHZ MIPS R10000 processors. Each processor has a separate 32 Kb first-level instruction and data cache, and a unified 4 Mb second-level cache with 2-way associativity using a 128-byte block size. The machine has 16 Gb of main memory (512 Mb per node) with a page size of 16 Kbytes. Each pair of nodes is connected to a network router.

2.3 Traditional scheduling policies

In a single processor system, the job scheduling is done in just one dimension. The only decision to be taken is which job execute next. In a multiprocessor system, the job scheduling has to be done in two dimensions. It is thus necessary to decide which job is scheduled next and also which process will be assigned to it and on which processor it will run.

Processor scheduling in a parallel processing context usually refers to process to processor mapping. This can be assumed when the pool of processors assigned to a job is for its exclusive use. But in case where there are more processes than processors in the system, they will have to be shared, so scheduling will also involve the sharing of these.

2.3.1 Processor scheduling

Scheduling algorithms can be divided into two main classes: *time-sharing* and *space-sharing*. [Feit97]. Time-sharing algorithms multiplex the time on a processor into several discrete intervals or slots. These slots are then assigned to unique jobs. In this way several jobs can share the same computer resource. At the other end, *space-sharing* algorithms assign the requested resources to a single job until the job completes execution. Most clusters operate in *space-sharing* mode. Due to the fact that both approximations are orthogonal to each other, it is possible to combine them in different ways. The best known of these is *gang-scheduling*.

2.3.1.1 Time-sharing algorithms

When one has more processes than processors, time-sharing algorithms multiplex the use of processors amongst jobs in time. This approach can result in good performance when executing sequential jobs as it reduces the average response time. However, it degrades the performance of parallel jobs because they are composed of processes that periodically synchronize, using a pure time-sharing approach, the periods of idle time of parallel jobs are significantly increased through lack of synchronization. In addition job processes must perform context switches very often when sharing the assigned processor.

The decision as to how to organize processes in queues, depends mostly on the memory architecture under consideration. If shared-memory is not available a global process queue is something difficult to implement.

Local queues are a natural approach for machines with distributed memory. Data locality is preserved; a process will always execute on the same processor. The main consideration here is how to do the process mapping to processors taking into account the load balancing.

Using a global queue, a process is chosen from it and then assigned to a given processor. Load balancing is done automatically. The disadvantages of using this approach are the creation of queue contention and the loss of data locality. The criterion to select the next process to execute from the global queue is usually based on priorities that take into account the use of the processor. Another possible criterion could be based on *affinity scheduling*, that is to say, to try and execute each process in the processor where it has been executed most recently. Nevertheless it is possible that the *cache* has not kept anything of the data from its previous execution due to the other processes that have will have executed after it, so this is not a good straregy to use.

2.3.1.2 Space-sharing algorithms

Space-sharing reduces the context switching effect, by partitioning the set of available processors amonst the jobs. The partitioning can be either static or dynamic. In static partitioning the set of assigned processors is fixed during the whole job's execution while in dynamic partitioning, processors are reallocated while the job executes depending on its requirements.

In these kinds of approaches the operating system is involved solely in the allocation of processes to processors as opposed to processor scheduling. This occurs more commonly in shared-memory multiprocessors.

The partitions can be static or dynamic. In the first case the partitions are predefined. Each application will be able to begin its execution if there is an available partition that satisfies the minimum requirements that it has asked for. It is a simple strategy; each processor is dedicated to run on just one process, so data locality is then preserved. But since the size of the partitions can not adapt exactly to the number of processes of the application, fragmentation could be generated. That is to say, resources that are not being used could remain idle or that there is no job in the wait queue that can adapts to the partitions available.

The static partitioning can be used to dedicate certain partitions to user groups or job classes, for instance a partition for batch jobs and another one for interactive jobs.

In dynamic partitions, the size changes dynamically reflecting any changes in the load or as a result of the requirements of its applications. The changes in the load can be due to new job arrivals or terminations. The changes in applications have to do with the amount of parallelism that is being applied at every moment. For example, when an application enters a sequential phase of calculation, it will release all the processors assigned to it except one. When it enters a parallel phase again, it must be assigned enough processors so that it can continue its execution. A disadvantage of this approach is that it does not preserve the data locality of its data and its flexibility depends on the programming model being applied in each application.

2.3.1.3 Mixed algorithms: Gang scheduling

In parallel multiprogrammed systems, processes are organized in gangs belonging to different applications. The processes of a parallel application cooperate with each other, competing with those of other applications. In *gang scheduling*, information about which processes of the same application cooperate with each other is known in advance. They are assigned to different processors and scheduled at the same time. Processors are shared in time between processes belonging to the different gangs.

In this way an application has the illusion that it is being executed in an exclusive form in the set of processors that it has been assigned, eliminating the synchronization problem.

The main disadvantage of this approach is that whenever a new gang of communicating processes executes, it is necessary to make a global context switch, which is not scalable at all. The context switches have to be synchronized by a central process, generating overhead and contention when maintaining global information. On the other hand the information about gangs must be provided before the beginning of the application's execution to make the appropriate scheduling.

2.3.2 Job scheduling algorithms

Whenever a work arrives at a system, it waits in a queue until the job scheduler decides to dispatch it. The queueing time for a job depends on many factors like the system load, the resource availability, the job scheduling policy. In this work, the response time of a job is the elapsed time from the point the job arrives at the system until its execution is completed.

Usually there are different job queues, each with a different priority [ZFMS00][SKSS02][IBSP04]. This makes the scheduling more complex. In addition more information is needed for each job, such as: the job size, the priority, the required execution time, etc. Much of this information is often provided by the user who does the job submission. In this context scheduling algorithms are necessary to ensure a good system performance.

Some examples of traditional job scheduling algorithms are: *First-Come-First-Served* (FCFS), *First-In-First-Out* (FIFO), *Shortest-Job-First* (SJF), *Longest-Job-First* (LJF). These algorithms can be improved if combined with *Backfilling* [Lifk94] techniques, taking advantage of the information of the execution times provided by the user.

2.3.2.1 *FCFS and FIFO*

FCFS and FIFO are the simplest job scheduling algorithms. Jobs are dispatched in the same order in which they arrive at the system. For the FIFO case no other job is dispatched until a job finishes its execution.

This scheduling algorithm does not take into account job priorities nor does it take advantage of any characteristic of the jobs. On the other hand they are very easy to implement.

2.3.2.2 *SJF* and *LJF*

In SJT and LJF the information about the execution time of each job is needed in advance. The job wait queue is reordered periodically according to this parameter, keeping the shortest (SJF) or longest (LJF) job first in the queue.

In SJF short jobs obtain a better response time, but long ones can suffer indefinite delays. The LJF algorithm tends to maximize the system utilization at the cost of the response times of jobs.

2.3.2.3 *Backfilling techniques*

The backfilling techniques first developed by [Lifk94] were proposed to improve system utilization and has been implemented in several production schedulers [JaSC01]. This technique is greatly known to increase user satisfaction since small jobs tend to get through faster, while bypassing larger ones.

This scheme tries to allocate short jobs in the gaps generated because of fragmentation without delaying any job in the wait queue. In order to apply backfilling it is necessary to know the execution time of jobs.

2.3.3 Job schedulers implementations

In this section the characteristics of some commercial and open source, of well-known job schedulers are commented on.

Loadleveler [laodl06] from *IBM* is a queue batch system which is in charge of executing parallel and sequential jobs. The requirements of each job are provided by the user at submission time. Job classes are defined depending on their maximum execution time. *Loadleveler* can do job *checkpointing* to continue its execution later. It is possible to ask for exclusive use of a set of processors. The *Loadleveler* has made certain functions accessible to users by means of APIs. These provide functions and control structures to deal with things such as manual *checkpointing.*

It is possible to configure external resource managers such as *Maui* [Maui], an open source scheduler. This one is in charge of doing the job scheduling and the resource allocation. It has the following scheduling policies: by priorities, *backfilling*, *throttling*, and *QoS*. The policies *throttling* tries to avoid monopolization of the resources from a group, a user, QoS or a queue, to accomplish this it assigns limits to their utilization. The policies of QoS allow the classification of jobs by classes, users, and groups.

At an academic level in [FPFC02] they present STORM, a package which is in charge of resource managing, job dispatching, and also manipulates efficiently communication between processes taking advantage of hardware facilities for short messages. This administrator consists of three types of daemons which perform each of the tasks mentioned above. They are synchronized through messages that make global context switches, or that interchange information about the state of the system.

2.3.4 Process mapping to processors

The initial allocation of processes is often referred to as *mapping*. After being assigned depending on the processor scheduling policy, processes can migrate or continue with their allocation for the remainder of their execution.

If processes are organized in local queues per processor, the initial placement will have a strong effect in the performance of the remainder of its execution. If processes are organized in global queues, they will be migrating continuously, so initial placement lacks importance.

One simple approach to do a mapping is to use a random placement. Each process chooses a processor at random and then is mappped it to that processor [KlPa84], [AtSe88].

This scheme could result in the most highly loaded processor always being chosen. In order to avoid this, the scheme can be applied in two steps: first a processor is chosen at random and then the processor that has the least load in its immediate neighbour is selected [GrNR90].

When working with parallel applications, where processes communicate frequently with each other, the mapping acquires greater relevance. Information about precedence and communicating processes is needed. However this information is not always available.

In [RoRi02] they propose two algorithms to map parallel applications to processors in a static way based on information, such as the data dependence, which must be provided in advance. They make their evaluations with synthetic applications that use message passing (PVM) to communicate and an image processing application named BASIZ.

2.4 Parallel programming models

The existence of parallel programming models has allowed the improvement of parallel processing through the programming of applications that take advantage efficiently of the parallel architectures.

This section presents some alternatives for parallel programming based on models with message passing like MPI [mpi] and PVM [pvm], which can be used on distributed and shared memory architectures, models for distributed shared-memory like UPC and models for shared-memory like OpenMP [openMP06].

2.4.1 Message passing models

Message passing is based on two primitives: send and receive. These functions involve concepts such as buffering (temporary storage), blocking or asynchronous modes and reliable communication. A message can be stored temporarily by the message subsystem in the source, the target or both. This will determine if this operation will continue its execution immediately or if it must be blocked until the message is stored temporarily (asynchronous) or until it is effectively received by the target (blocking).

Communication is reliable if the messages are guaranteed to arrive at the target, if the order is preserved and if there is any emergency plan in case a message is corrupted.

The characteristics of the two most widely used message passing models: MPI which is used in this work and PVM are described next.

2.4.1.1 Message Passing Interface (MPI)

MPI is a specification that defines the semantics of a set of functions that form an interface that allows communications between processes to take place. MPI was proposed as a standard by a committee composed of developers, users and vendors. MPI does not define the protocol that implements these operations (i.e. whether TCP/IP sockets must be used) as it does not specify how it must be implemented. In the creation of this specification, the most interesting characteristics of already existing communication libraries were considered such as [BaKi92], [Pier88].

The first version of the specification was written by Dongarra, Hempel, Hey, and Walker appeared in November of 1992, and one reviewed complete version appeared in February of 1993 [DHHW93].

Working with a message passing paradigm requires a low level approach. This means that the parallelism must be expressed in an explicit form. The work and the data have to be distributed in an explicitly way between the processes of the parallel application and the communication is done through messages.

The object of MPI was to develop a standard interface that would be widely used to write programs that used message passing to communicate. Such an interface must be practical, efficient and flexible.

Advantages of using MPI

The advantages of using the MPI library over other message passing libraries is the portability it offers, as MPI has been implemented for most any architectures, using distributed and shared memory.

Functionality

The MPI interface provides synchronization and communication functions that act between the processes in a language independent way of the language, with a specific syntax.

These functions include point-to-point communication operations with blocking and non-blocking variants (send and receive), reduction operations (collective sum, maximum value, minimum value), gather/scatter, global synchronizations (barrier) as well as operations to obtain system information like the number of processes, the processor to which a process is currently mapped to.

Figure 2.1, Figure 2.2 and Figure 2.3 [Zeeh04] show graphically the behaviour of three commonly used MPI collective. The *MPI_barrier* function shown in Figure 2.1 provides a *rendez-vous* point for the processes participating in the execution. The barrier function returns when all the processes in the group have executed the operation. Tipically, barriers are used to separate different calculation phases in the execution, for example just before exchanging intermediate results. The *MPI_bcast* function, schematically shown in Figure 2.2, delivers a message from the invoking process to the rest of the processes in the same group. The *MPI_alltoall* function shown in Figure 2.3, delivers data from the invoking process to the rest of the processes that belongs to the same group and gets data back from them. This function returns once the deliver and the gather are completed.

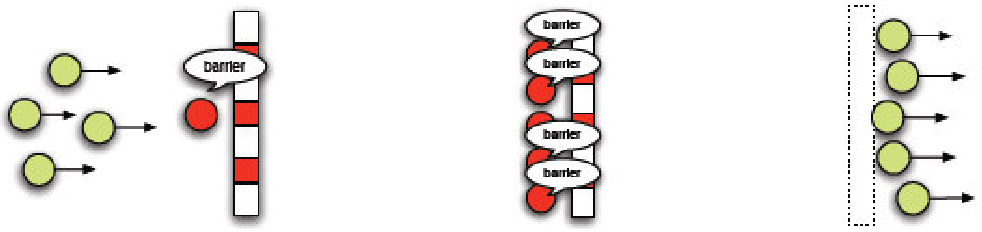


Figure 2.1 Global synchronization (MPI_barrier)

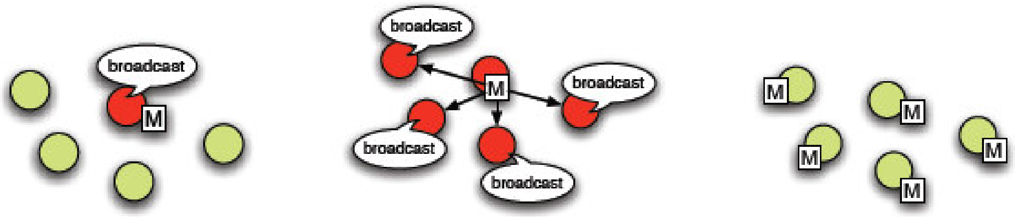


Figure 2.2 Broadcasting (MPI_bcast)

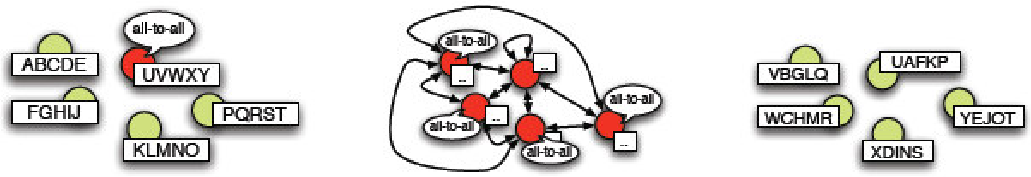


Figure 2.3 Scatter and gather (MPI_alltoall)

Implementations

Most of the implementations of MPI are made available in a library that consists of a set of the routines (API) that are linked to programs written in FORTRAN, C or C++ and, by extension, by any language that is able to support an interface with the routines in the library.

The standardization process was characterized by the cooperation of vendors and researchers, and made available through commercial implementations, such as those provided by Sun, SGI, IBM, and through open source implementations like those provided from MPICH [mpich], LAN/MPI [lanmpi] and Open MPI [OpenMP05].

MPICH

MPICH [mpich] is an implementation of MPI of open source code that includes platforms such as clusters of SMPs and MPPs. The "CH" in MPICH comes from Chamaleon (chameleon), referencing its adaptability and portability. It was created with the aim of providing an environment for the development of new and better environments for parallel programming.

The idea was to allow to the MPI community of users and researchers to evaluate the viability of their ideas. Within the supported platforms there are Unix and Windows NT flavours of MPICH.

LAM/MPI

LAM (Local Multicomputer Area) [lanmpi] is an open source MPI programming environment for a network of heterogeneous machines.

With LAM/MPI, a set of machines constituted in a network can act as a single resource for parallel computation.

It was developed at the Ohio Supercomputing Centre and is maintained by the University of Notre Dame and the University of Indiana. It supports most of the POSIX platforms. LAM/MPI uses demons for their runtime environment. The execution and remote authentication are based on the well established *rsh* and *ssh* programs.

As the runtime environment of LAM is started independently from a LAM/MPI application, launching LAM/MPI applications is usually faster than when other MPI implementations are used.

Open MPI

Open MPI [OpenMP05] has a growing community based on an MPI open source implementation, which combines technologies and resources from previous projects such as LAM/MPI, in order to construct an improved MPI library. It was developed by a collaboration of research centres and vendors. The objective was to create an open source implementation, which had a high performance on a heterogeneous network, with the agreement of vendors for its standardization and that supported a wide range of platforms and development environments.

The performance of Open MPI at a hardware communications level still needs some optimization [openmpi05]. In this sense, its performance is still below that of LAM/MPI and MPICH. There exist characteristics which have not been ported from LAM/MPI to Open MPI yet, even though it was constructed using the best ideas of LAM/MPI. It is expected to be improved in the near future, as Open MPI is trying to be a worthy successor.

2.4.1.2 Parallel Virtual Machine (PVM)

The Virtual Parallel machine (PVM) [pvm] is a software package that allows an abstraction of a heterogeneous network of Unix and/or Windows machines to be created, the virtual parallel machine, which appears to an application as a single parallel resource.

PVM was developed by the University of Tenesse, Oak Ridge National Laboratory and Emory University. The first version was made available in 1989. The advantage of PVM is that it allows exploiting existing hardware to be exploited to solve problems with high computational requirements, at a relatively low cost.

Under this model, the programmer writes an application as a set of cooperating tasks which access the PVM resources through a standard interface. These routines allow the initiation and completion of tasks through the network, as well as the communication and synchronization of them. It uses message passing for the communication and exchange of information. Communications can be done in a point-to-point manner, broadcasting, using global synchronizations and/or global sums.

The PVM software relies on several daemon processes that are run at each node and are replicated for each user of the system. The idea is that these processes make the addressing across the nodes and increase the security when sharing software between users. In order to obtain location transparency and fault tolerance it uses a global knowledge strategy between the daemons and identifies instances using global identifiers. Nevertheless all this entails an extra overhead for the system.

MPI and PVM are both designed to provide the user with a library to write portable and heterogeneous codes [GrLu02]. MPI has a bigger set of communication primitives than PVM. For this reason an application with special communication requirements would be better off choosing MPI. The most cited example of the communications difference is the asynchronous send which MPI has and PVM does not. On the other hand, there are incompatibilities between the different MPI implementations, i.e. it is not possible to communicate between different MPI implementations. PVM has a protocol to recover from failures, for example if a node crashes. But this type of failure [GeKP96] requires previous notification.

2.4.2 *Unified Parallel C (UPC)*

Unified Parallel C (*UPC*) [upc] is an extension of the C programming language designed to get high performance computing on parallel machines with shared-memory address spaces, such as SMPs, or with distributed memory like clusters.

The system appears to the programmer as a single address space, where the variables can be read and written directly from any processor. The language provides constructions that allows shared data or distributed shared data to be declared, to synchronize threads and to share data between threads. UPC combines the advantages of programming on shared-memory and the advantages of using a message passing programming model.

In order to express parallelism, UPC extends ISO C 99 [ISOC99] with an explicit model of parallel execution, a shared address space, synchronization primitives and a consistency memory model, and finally some primitives to handle memory operations.

There exist some commercial implementations; the first one was *HPC UPC* [hpcupc00] which appeared in December of 2000. There are also open source implementations like *Berkerley UPC* [Berk06] and IBM [IBMU][BCAY06].

The performance of applications written using the UPC library for fine grain algorithms in distributed environments is poor. For applications of coarse grainularity it has a similar performance to MPI. Nevertheless, as the communication becomes more complex, MPI obtains a better performance, on clusters and on shared-memory. On the other hand UPC does not have collective operations which can significantly complicate the code [Berl02] [BHJK04].

2.4.3 *OpenMP*

OpenMP (Open Multi-Processing) [openmp] is an application programming interface (API) for shared-memory systems for programs written in Fortran, C and C++. It consists of a set of compiler directives, library functions and environment variables which control the behavior of the application at runtime. It is a portable model and is supported by HP, IBM, Intel, SGI, Sun and others on Unix and NT.

The committee that defines the standard, the *OpenMP Architecture Review Board* (*ARB*), published the first OpenMP standard for *Fortran 1.0* in October of 1997. The following year a standard for *C/C++* appeared.

The execution model for OpenMP is a fork-join one. A program begins its execution with a single thread named the master thread. The master thread is executed sequentially until it finds the first parallel construction (such as PARALLEL followed by an END PARALLEL directive). At this point the master thread creates a set of threads, including itself as part of the set. Each thread has its own data area but it can also share data by specifying this at the beginning of the parallel construction.

OpenMP does not have to deal with messages. By default data is shared except it is specified the contrary. Parallelism can work with a portion of the program and thus it is not necessary to make dramatic changes in the code. The same code can be executed in sequential and in a parallel form.

On the other hand this model can work only shared-memory machines. It also requires a compiler that supports this model. Parallelism is performed just at the loop level, leaving outside the code that potentially could be parallel but that it does not takes part of any loop.

2.5 Rigid, moldable and malleable jobs

According to [FRSS97] a work can be characterized as: *rigid*, *moldable* and *malleable*. A job is defined as being *rigid* when the number of processes used is specified in an external form to the scheduler and remains fixed during the whole execution. A job is said to be *moldable*, when a job can be executed on multiple processor partition sizes. Nevertheless, once the execution starts, these sizes cannot be modified. Finally a job is defined as being *malleable*, if the size of the assigned partition can also be modified during the execution. The impact of executing a job of each type of this classification can be observed in Figure 2.4.

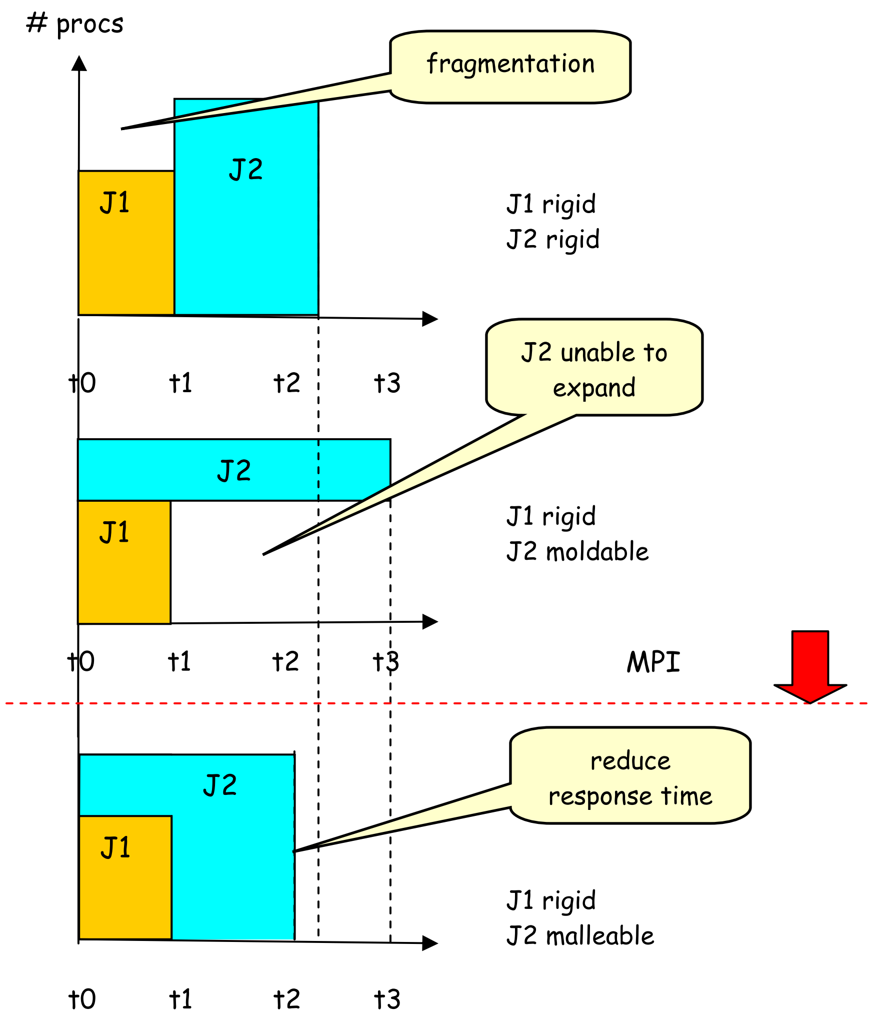


Figure 2.4 Impact of the execution of a rigid, a moldable and a malleable job.

The moldability reduces the waiting time as the sizes of the jobs are adapted to the resources available at the beginning of the execution. However, the size will remain fixed even though the load of the system varies. Only malleable jobs have the ability to adapt to such changes.

MPI jobs can be moldables but it does not support malleability. When the number of processes is varied, the data needs to be redistributed in an explicit form to redistribute the work. This requires extra effort from the programmer, since this functionality will have to be programmed explicitly.

2.5.1 Example of execution

In this section the execution characteristics for scheduling rigid and moldable jobs are analyzed. Figure 2.5, Figure 2.6 and Figure 2.7 show the execution of two jobs, J1 and J2. J1 has a smaller execution time than J2 and both arrive at the system at the same time.

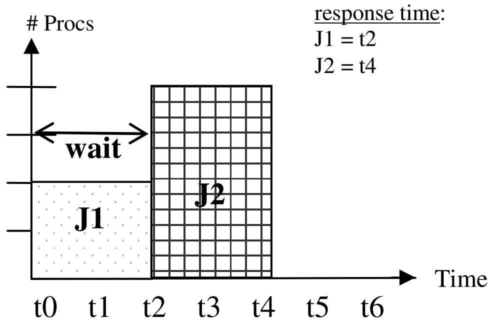


Figure 2.5 FCFS execution of two rigid jobs

Figure 2.5 shows the execution of *J1* and *J2*, both rigid jobs. *J1* asks for two processors, while *J2* asks for four. J1 began execution at *t0*, and J2 has to wait until *J1,* finishes its execution as there are not enough resources for it. It is possible to see the fragmentation generated from *t0* to *t2*, where two processors remained idle even though there was a job in the wait queue.

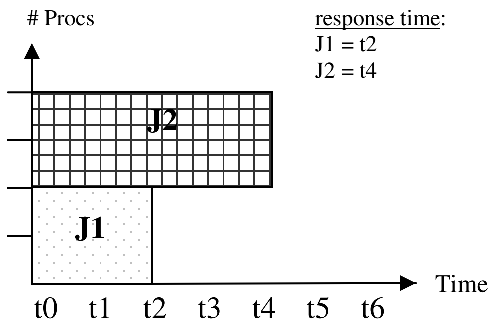


Figure 2.6 FCFS execution of two moldables jobs

Figure 2.6 shows the execution of two moldable jobs *J1* and *J2*. Both begin execution at *t0*. *J1* asked for two processors and *J2* was assigned the rest of the available processors, two processors. Although J1 finished its execution before J2 ended, J2 could not take advantage of the resources that have just been freed.

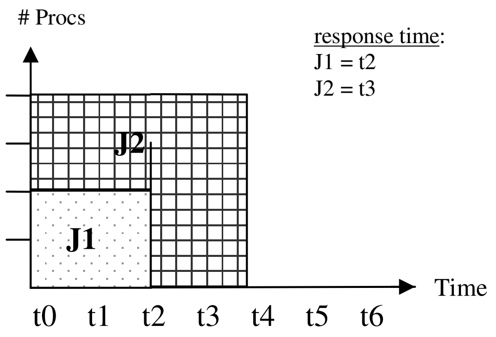


Figure 2.7 FCFS execution of two malleables jobs

Figure 2.7 shows the execution of two jobs, *J1* and *J2*, which are both malleable and begin execution at *t0*. J1 finishes its execution before *J2* ends, but as J2 is malleable and as soon as new resources become available; J1 can then expand and take advantage of them.

2.6 Summary

This chapter has presented the different elements that form part of a multiprocessor system.

First, the types of architecture that vary their memory organization were presented. Next, the main components of the operating system which are in charge of the resource management as well as the job scheduling were described.

Finally, there was a brief overview on parallel programming models with the most common commercial and open source code libraries being described. The programming model used in this work, the MPI, was presented.

Chapter 3
EXECUTION ENVIRONMENT

Abstract

*The performance evaluations of the contributions of this work were conducted using real executions. In order to do that an execution environment was developed. This environment is composed by a queuing system named **launcher**, which is in charge of the job scheduling, a resource manager named **cpu manager**, which is in charge of the processor allocation and a runtime library named **vmruntime**, which is in charge of doing the process mapping and process scheduling.*

This chapter presents the implementation and functionality of the elements above mentioned.

3.1 Introduction

This chapter describes the components of the execution environment used to evaluate the contributions of this work.

The execution environment is made up of a queuing system, *launcher*, which is in charge of the job scheduling, a resource manager, *CPUM Manager (CPUM),* which is in charge of the processor allocation and a runtime library, *VMruntime*, which take the process scheduling decisions.

Once the applications begin their execution are under the control of the *CPUM*, which along with the *VMruntime*, make the processor scheduling on the assigned partition. In order to achieve this objective the *VMruntime* intercepts the calls to any message passing library MPI function as well as a call to a system routine, the *sginap*. In this way the *CPUM* is able to control the applications in an external form, with no need of recompilation and transparently to the user.

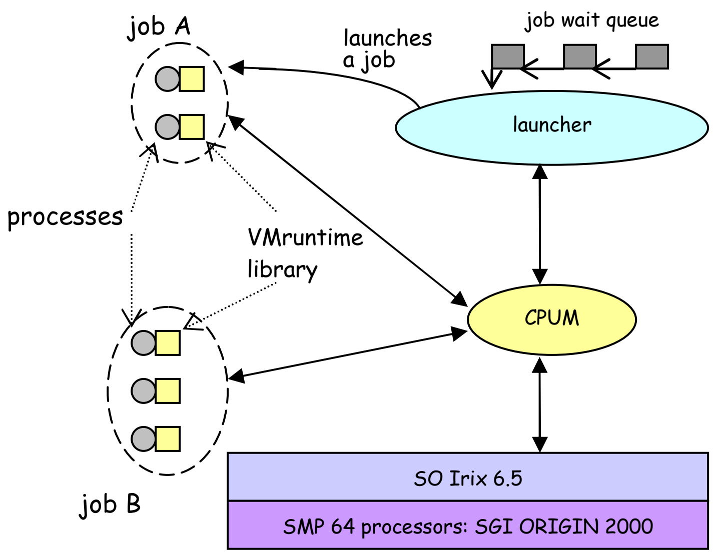


Figure 3.1 Relation between the components of the execution environeent

Figure 3.1 shows the main elements of the execution environment as well as the relation between them. Each time a job arrives to the system, the launcher allocates it in the wait queue. The launcher decides when to execute it, and in coordination with the *CPUM* it also decides the number of processors to assign to it, the number of processes and the maximum multiprogramming level (*MPL*) allowed for it.

The multiprogramming level of a job is defined in this work as the result of dividing the number of processes into the number of processors, rounded up.

The *CPUM* manages the resource assignment to applications. The *CPUM* communicates with the *VMruntime* library and the *launcher* through shared memory. This runtime library performs the process mapping and scheduling.

In order to implement the proposals of this work it was neither modified the operating system, nor the message passing library.

This chapter is organized as follows, in sections 3.2, 3.3 and 3.4 is described in detail the launcher and the *CPUM*. In section 3.5 is presented the *VMruntime* library used to intercept the calls to the message passing library MPI. In sections 3.6 there is an overview of the applications used for the evaluations, the workloads as well as some application classifications. In section 3.7 are described the formulas used to construct the traces for the workloads. Finally in section 3.8 is the summary of the chapter.

3.2 The job scheduler: Launcher

Queueing systems are an important tool in the evaluation of system performance. As in this work there are proposals at the job scheduling level, it was necessary to have an own job scheduler in order to implement them. This section is dedicated to the description of the queueing system used for the implementation of the work, the *launcher*.

The *launcher* in coordination with the *CPUM*, and applying the corresponding scheduling algorithm, selects one job from the wait queue. It decides when to launch the selected job, its number of processes and its maximum multiprogramming level.

The decisions mentioned before are closely related with the policies being applied at job scheduling level, which are part of the proposals of this work and are described in the following chapters.

In order to evaluate and compare the performance for a given workload under different scheduling policies, the *launcher* accepts as a parameter a workload trace file which specifies the arriving times of the jobs in the workload. Using this trace it is possible to execute the same job sequence with identical arriving times to the system in different experiments.

3.2.1 Parameter files

The *Launcher* accepts the following files as parameters:

1) A list of posible applications of the workload.

The list is defined in a template file, where each line corresponds to a different application. Each application is written with its absolute path and the necessary commands to execute it, like *mpirun*.

```
mpirun –np N $HOME/aplicaciones/bin/cg.B.N
mpirun –np N $HOME/aplicaciones/bin/bt.A.N
mpirun –np N $HOME/aplicaciones/bin/send.50x2
```

Figure 3.2 Example of a template file with the list of applications of a workload

The number of processes that an application would run is not specified in the command line. This number is parametrized and it is indicated by the letter "**N**". In the case of the NAS applications [Nas03], their names have also the character "**N**" instead of the number of processes. An example of this file is shown in Figure 3.2.

The *launcher* reads this file at the beginning of the execution of the workload. During the execution of a workload, and just before launching each application, it decides its number of processes and substitutes it in the command line.

When using the NAS it is not possible to specify different number of processes for the same application before running it. This number has to be specified during compilation time, which means that there exists a different binary file for each number of processes. The same happens with the sweep3D [sweep3d]. For this reason the binaries are also parametrized. This is not the case for the synthetic applications (i.e. *send.50x2*)

2) Number of processes allowed for each application:

```
32 16  8   4   2  1  1
60  -1 -1 -1 -1 -1  -1
49 36 25 16 4  4  4
```

Figure 3.3 Example of a file with the possible number of proceses that can be run

In Figure 3.3, it can be observed the number of processes allowed for each application. Each line of the file corresponds to an application in the same order of the file in Figure 3.2. Currently are allowed a maximum of seven different numbers for each application. To specify that an application admits any number of processes up to a maximum specified first, -1 is written in the rest of the columns.

3) Job type (long or short):

```
1
2
1
```

Figure 3.4 Example of a file with the job type for each application

Figure 3.4 shows an example of a file containing the job types for each application of the workload. Each line corresponds to an application in the same order of the file in Figure 3.2. The type of the job is indicated with a number, if it is short the number is 2, if it is long the number is 1. For this work were considered just two types. In case of admitting more possibilities, the rank of numbers should be extended.

4) Trace with the arrival times for each job in the workload

```
job_number  arrival_time        application_number

  0   2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2 -1 -1 -1 -1
  1 34 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2 -1 -1 -1 -1
  2 36 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1
  3 67 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1
```

Figure 3.5 Example of a file with the arrival times for a specific workload

The trace follows the *Standard Workload Format (SWF)* proposed by Feitelson in [FEIT97]. Figure 3.5 shows an example of a trace for a given workload. The second column of the trace corresponds to the arrival times of the applications expressed in seconds. The next column with a number not equal to -1, corresponds to the number of the specific application.

3.2.2 Job scheduling algorithm

This section describes the job scheduling algorithm implemented by the *launcher*. Figure 3.6 shows the pseudocode of such algorithm.

```
JOB SCHEDULING algorithm

  while launcher_running do
      if not_empty_wait_queue()
          job = first_job_wait_queue();
          ok= get_number_processes (SCHEDULING_POLICY, num_free_processors,
                                     &num_processes, &max_MPL);
          if not ok then
              if backfilling_allowed then
                  if expired_windows() then
                      apply_expired_windows (backfilling_policy);
                  else
                      job = backfill_job(&num_processes, &max_MPL);
                  end if
          end if
          if (job >= 0) then /* a valid job */
              change_job_status(job, RUNNING);
              update_wait _queue();
              assign_number_of_processes (job, num_processes);
              assign_maximum_mpl (job, max_MPL);
              recalculate_number_of_free_processors();
              launch (job);
          end if
      end if
  end while

END algorithm
```

Figure 3.6 The *launcher* job scheduling algorithm

The job scheduling algorithm consists of an infinite loop. At each iteration an attempt is made to dispatch the first job in the queue. In order to do that, given the number of free processors and the job scheduling policy currently applied (chapter 5), a number of processes and a maximum multiprogramming level (*MPL*) is selected. Depending on the job scheduling policy and job characteristics the algorithm treats the job as moldable or rigid.

If it were posible to find a combination of *MPL* and a number of processes that satisfies the job requirements in combination with the availability of resources in the system, then the job is executed. If it were not the case and *backfilling* was allowed, the launcher tries to free resources from backfilled jobs which have expired their window execution time. Finally if that was not even possible but *backfilling* was allowed, the launcher tries to backfill a job from the wait queue that could adapt to the available resources.

To execute a job, the *launcher* updates the job status to *RUNNING*, deleting it from the wait queue, and assigns to it the selected number of processes and the maximum multiprogramming level. The number of free processors is recalculated.

3.3 *Resource manager: CPU Manager (CPUM)*

The *CPU Manager* or *CPUM* is a processor scheduler implemented at user level. Once the *launcher* dispatches a job for execution, the job enters under the control of the *CPUM*. The *CPUM* is in charge of the processor allocation. The communication between the *CPUM* and the jobs is done through shared memory, by means of control structures. The *CPUM* uses the native interface of the operating system to apply the scheduling policies. It was constructed on the top of IRIX 6.5, which is the native operating system of SGI Origin 2000.

In order to get control of *MPI* jobs, a dynamic interposition mechanism is used. Through this mechanism, all the calls to the *MPI* library are are intercepted. For a more detailed description refer to the next section. All this functions provide information to the *CPUM* or get information from it.

All the scheduling policies were implemented external to the *MPI* library and without the need of recompilation.

There exist also stored information about the jobs and the system state. They are used to keep information for the scheduling decisions. This information consists of statistical information, timings, the internal MPI identifier and information related to the communication between the processes of an application.

3.3.1 *CPUM* functionality

The *CPUM* wakes up periodically at each quantum time expiration. It examines the new arrived jobs and the ones which have just finished, updating the control structures. It redistributes processors and depending on the currently scheduling policy being applied (i.e. periodic boost) it makes the necessary context switches between processes.

The *CPUM* is in charge of the processor allocation. Once an application is assigned a number of processes and the *MPL*, the *CPUM* does:

- Calculate the processor partition size for each application
- Which processors assign to each application
- When to make the global context switches among process. This happens just when some specific scheduling policies like the *Periodic Boost* [ZSMF00] are applied. For a more detailed description refer to chapter 4.

In order to be able to carry out these tasks the *CPUM* has private data structures and shared data structures which are accessed also by the *VMruntime* library and the *launcher.* Those shared data structures can be updated by any of them.

3.3.1.1 Command line parameters

The *CPUM* accepts the following parameters in the command line:

- Total number of processors in the system
- Processor allocation policy (chapter 5): FJT, PSA, ASP, Folding, FCFS.
- Processor sharing (chapter 4): time-sharing, space-sharing, FIFO.

- Synchronization policy (chapter 4): spin blocking, blocking immediately, busy waiting.
- if *Backfilling* techniques are allowed or not (chapter 6) and window expiring policy: abort, malleability
- The processor number where the *CPUM* is attached to. This parameter is to force the execution of the *CPUM* outside the set of processors dedicated to the execution of applications. In this way, its execution won't affect the performance results.
- Maximum multiprogramming level allowed for all the applications in the system.
- *Spin time*, which is the maximum time that a process will wait for a message before blocking. This is used when applying coscheduling policies (chapter 4).

3.3.1.2 Processor allocation

The number of processes in the system can be greater than the number of processors, so processes have to compete for the use of processors. Depending on the processor sharing policy that is being applied, each processor is shared by processes from different applications or by processes from the same application.

When time-sharing policies are applied, the applications get as much processors as they ask to. However, they must time share the resources between them. It means that processes from different applications will compete for the use of the resources.

When space-sharing policies are applied, as the number of processes could be greater than the assigned partition size, even the processes have to share the processors; they all belong to the same application. The partition size depends on the maximum the number of processes and the multiprogramming level (MPL) applied to the application.

Figure 3.7 shows graphically how processors are shared between processes when time-sharing is applied (left) and when space-sharing is applied (right). Each vertical column of circles represent processes allocated to that processor (grey square). It can be observed that processes from the same application are allocated in different processors, when applying time-sharing, so they must time share with processes from other applications. On the other hand in space-sharing, an application is allocated in a processor partition for its exclusive use. In this way the processes compete with themselves for the use of the resources.

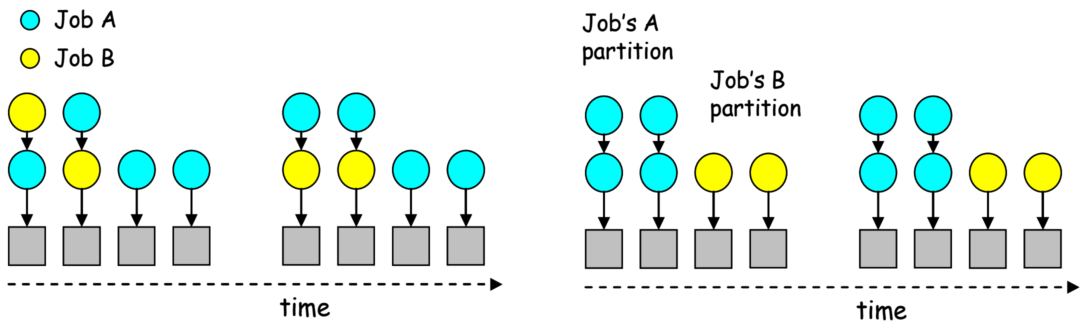


Figure 3.7 Processes allocation when applying time-sharing (left) and space-sharing (right)

This calculation of the partition size for each application is made separately and is recalculted each time a change in the system conditions happens, like the arrival of new jobs or the termination of others.

The resources are distributed equitatively and proportional to the number of processes of each application. Each processor is assigned the same number of processes within an application..

$$\text{partition size} = \frac{\text{\# processes of the application}}{\text{MPL}}$$

Equation 3.1 Calculation of the processor partition size

If the result from the calculation of Equation 3.1 were not an exact number, then the rest of processors are redistributed among applications trying to keep the same number of processes per processor. If that is not even possible, then they are redistributed among applications in an equipartition way, if and only if there aren't jobs in the wait queue.

Memory affinity is tried to keep, so once a set of processors are assigned to an application, they remain attached to it unless the partition size is modified. In case this size is reduced, the application looses part of the processors. Any previous processor assignment is not taken into account in future allocations.

3.3.2 Coordination between the *CPUM* and the *Launcher*

Whenever a work arrives at the system, the *launcher* places it in the wait queue, decides when to execute it and takes scheduling decisions that affect the whole execution of the application, such as its number of processes.

All the decisions are based on information obtained from the system or from the *CPUM*. Since the *CPUM* manages the resources of the system, it can provide information about the availability of them.

As can be observed in Figure 3.8, the *CPUM* and the *launcher* exchange information through shared memory, using the data structures commented in section 3.3. The *CPUM* ask the *launcher* through a *named pipe*, for a new job every time there are available resources.

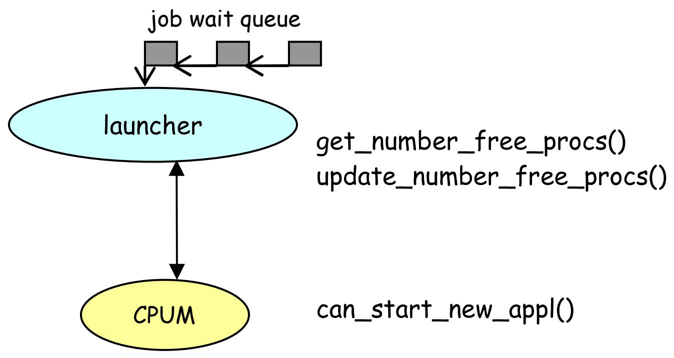


Figure 3.8 Coordination between the *launcher* and the *CPUM*

3.4 VMruntime library

In order to get control of the *MPI* jobs, a dynamic interposition mechanism is used. In particular it is applied the DiTools Library [SeNC00] that allows applications to intercept functions like the *MPI* calls and a system call routine: the s*ginap.* This routine is invoked by any function of the *MPI* library whenever they perform a blocking function.

All these intercepted functions provide information to the *CPUM* or get information from it. In addition using this mechanism it is possible to inhibit the execution of the *sginap* routine. The *sginap* wrapper, implemented as part of the VMruntime library, is in charge of doing the context switching each time the *spin time* has expired and decides which process goes next.

The interposition mechanism was used also to initialise the control structures of the *VMruntime library*, to find out each process MPI rank and to trace the MPI unconsumed messages in order to implement priorities.

This *VMruntime library* allows the implementation of the scheduling policies at process level. Each process can access information from all the processes of the application through the shared data structures which are described in section 3.6.2.

The scheduling policies implemented at this level define:

- Synchronization between the communicating processes of an application. This involves deciding:
  - When to yield the processor
  - Which process goes next
- From which queue (local or global) select the next process to execute.
- Process mapping to processors

Information about messaging and process statistics like the number of context switches, the number of messages sent, messages received, is also updated.

3.4.1.1 Process mapping to processors

The processors assigned to an application remain attached to it provided the partition does not undergo changes. The allocation is kept fixed whenever it is possible to minimize cache faults.

When working with processes organized in global queues even processes are assigned to a processor partition, they are not attached to them. Processes can make migrations between the processors of the same partition.

Process mapping has relevance when applying processes local queues per processor, as processes remain attached to processors during the whole execution.

The mapping algorithm applied in this work is very simple; it does not require any extra information. In chapter 2 it was discussed several processes mapping algorithms. Some of them take into account information about the application like the communication pattern or any known imbalanced behaviour. These kinds of algorithms could improve the execution time of the applications, mainly when incrementing the MPL, but this is out of the scope of this work.

Next is described the process mapping done in this work.

Process mapping used in this work

The mapping algorithm used in this work assigns the first process to the first processor of the partition, and so on in round robin. This is accomplished following the internal MPI identifier. The mapping is not random and it is ensured that for different executions and equal partition sizes, the conditions are identical, expecting in this way almost the same performance between executions.

In Figure 3.9 can be observed an example of the mapping algorithm applied in this dissertation. The first processor from the assigned partition, *P1,* is assigned process 0, then *P2* is assigned process 1, *P3* is assigned process 2. When the processors finish, the algorithm begins again by first of the list, in the example *P1* is assigned process 3 and so on.

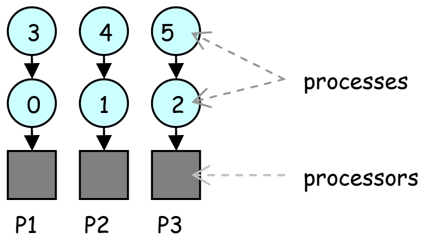


Figure 3.9 Process mapping following the *mpi_rank*

3.5 Data structures accessed by the CPUM, the launcher and the VMruntime library

There are private and shared data structures which are accessed and updated by the *CPUM,* the *launcher,* and the *VMruntime* library. They have information about the jobs in execution and in the wait queue as well as information about their processes and the processor allocation.

3.5.1 Data structures accessed by the *launcher* and the *CPUM*

As already mentioned in section 3.4.2 the *launcher* and the *CPUM* coordinate through a named pipe. The *CPUM* writes in it every time there are free resources. The *launcher* applies the job scheduling algorithm described in Figure 3.19. If there was a matching between the availability of the resources and any job size according to the current job scheduling policy, then this job is selected for execution.

In order to take all the decisions, the *launcher* keeps information about the jobs in the wait queue. The *CPUM* must know the number of processes assigned to each application as well as its maximum multiprogramming level.

So the *CPUM* and the launcher share:

- A list of running or finished jobs, where for each one it is known:
    - MPL
    - Job status: *RUNNING, FINISHED*
    - Number of processes
- Total number of processors in the system
- Total number of free processors

On the other hand the launcher keeps private information for its scheduling decisions. This information includes also the jobs that are still waiting in the queue:

- A list of jobs, where for each one it is known:
    - Name
    - Time spent in the wait queue
    - Job type (long or short)
    - Job status: *WAITING, RUNNING, FINISHED*
    - Arriving order
    - Execution order, if it applies

3.5.2 Data structures accessed by the *CPUM* and the *VMruntime* library

The *CPUM* is in charge of doing the processor allocation, so in order to do the process mapping, the VMruntime library must know the assigned partition.

For this reason the *CPUM* and the *VMruntime library* share a data structure. This structure has the following information for each application:

- List of processors assigned
- Partition size
- MPL

The *VMruntime library* keeps information at processor level and at application level. This information is used by the runtime library to schedule the processes queue and to do the process mapping to processors.

Processes can be organized in two possible ways: in local queue per processor or in a global queue per application. Depending on the processor scheduling policy currently being applied, the library has to choose the next process to execute from a local queue or from a global queue. For a detailed description about the queue types refer to chapter 4.

The data structure at processor level is accessed by every application in the system and has the following information per processor:

- A list of processes assigned to it
- A pointer to the currently executing process
- Number of currently assigned processes
- The process that thas last executed on it. This information is interesting for global queues when performing affinity scheduling.

Each running application has its own data structure with the following information:

- The list of processes with:
  - Pid
  - Processor where it has last executed
  - MPI internal identifier (*MPI_rank*)
  - number of unconsumed messages
  - number of sent messages and not consumed yet
- Time at it has begun execution
- Time at it has finished execution, if it applies
- Total number of processes

The messages that were sent and not yet consumed are the ones sent but the target process hasn't arrived to the point of execution where they are actually received, i.e. the execution of the MPI function *MPI_recv()*. This can be due to the fact that the target process is not currently being executed or it is performing a previous blocking function (i.e. *MPI_barrier*()). The unconsumed messages, are the ones which have been received but the current process hasn't performed yet the corresponding receiving function due to the reasons before explained.

3.6 Applications and workload design

In this section are described each one of the applications used for the performance evaluations of this work. It is also described how were generated the workloads that took part of the evaluations.

3.6.1 Applications used

For the evaluations the MPI NAS *benchmarks* [NAS03] [BHSW95] were considered including the multizone version of the *bt*, the *Sweep3D* [Sweep] and some synthetic applications.

The synthetic applications were used for the study of load balancing between the processes of an application. In this way, the load balancing degree could be manipulated, to make the necessary measurements.

3.6.1.1 Description

This section describes the applications involved in the evaluations. Firstly are analyzed the MPI NAS *benchmarks*. For each one there are up to five problem sizes, these are related with the data volume they manipulate (S, W, A, B, C). For a detailed description refer to [Nas03]. Next there is a brief description of each one:

EP: The first of the five kernel benchmarks is an *embarrassingly parallel* problem. In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudorandom numbers, which are generated according to a particular scheme that is well-suited for parallel computation. This problem is typical of many *Monte Carlo* applications. Since it requires almost no communication, in some sense this benchmark provides an estimate of the upper achievable limits for floating-point performance on a particular system.

MG: The second kernel benchmark is a simplified multigrid kernel, which solves a 3-D Poisson PDE. This problem is simplified in the sense that it has constant rather than variable coefficients as in a more realistic application. This code is a good test of both short and long distance highly structured communication. The Class B problem uses the same size grid as of Class A but a greater number of inner loop iterations.

CG: In this benchmark, a conjugate gradient method is used to compute an approximation to thesmallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication and employs sparse matrix-vector multiplication.

FT: In this benchmark a 3-D partial differential equation is solved using FFTs. This kernel performs the essence of many *spectral methods*. It is a good test of long-distance communication performance. The rules of the NPB specify that assembly-coded, library routines may be used to perform matrix multiplication and one-dimensional, two-dimensional, or three-dimensional FFTs. Thus this benchmark is somewhat unique in that computational library routines may be legally employed.

LU: The first of these is the so-named the lower-upper diagonal (LU) benchmark. It does not perform a LU factorization but instead employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block *5x5* lower and upper triangular system. This problem represents the computations associated with a newer class of implicit CFD algorithms, typified at NASA Ames by the code *INS3D-LU*. This problem exhibits a somewhat limited amount of parallelism compared to the next two benchmarks. A complete solution of the LU benchmark requires 250 iterations.

SP: The second simulated CFD application is named the scalar pentadiagonal (SP) benchmark. In this benchmark, multiple independent systems of nondiagonally dominant, scalar pentadiagonal equations are solved. A complete solution of the SP benchmark requires 400 iterations.

BT: The third simulated CFD application is named the block tridiagonal (BT) benchmark. In this benchmark, multiple independent systems of non-diagonally dominant, block tridiagonal equations with a *5x5* block size are solved. SP and BT are representative of computations associated with the implicit operators of CFD codes such as *ARC3D* at NASA Ames. SP and BT are similar in many respects, but there is a fundamental difference with respect to the communication to computation ratio. Timings are cited as complete run times, in seconds, as with the other benchmarks. For the BT benchmark, 200 iterations are required.

BT-MZ: The number of zones in this benchmark grows with the problem size in the same fashion as in SP-MZ. However, the overall mesh is now partitioned such that the sizes of the zones span a significant range.This is accomplished by increasing sizes of successize zones in a particular coordinate direction in a roughly geometric fashion. Except for class S, the ratio of largest over smallest total zone size is approximately 20.

Another *benchmark* used, which does not take part of the NAS is:

Sweep3D: Represents the heart of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by an IJK logically rectangular grid of cells. The angular dependence is handled by discrete angles with a spherical harmonics treatment for the scattering source. The solution involves two steps: the streaming operator is solved by sweeps for each angle and the scattering operator is solved iteratively.

Finaly the synthetic applications used are described:

send: Synthetic applications consist on a loop with three phases: communication, calculation and global synchronization. The communication is made between processes with even and odd identifiers. The imbalanced is forced by varying the amount of calculation.

In this way it is obtained:

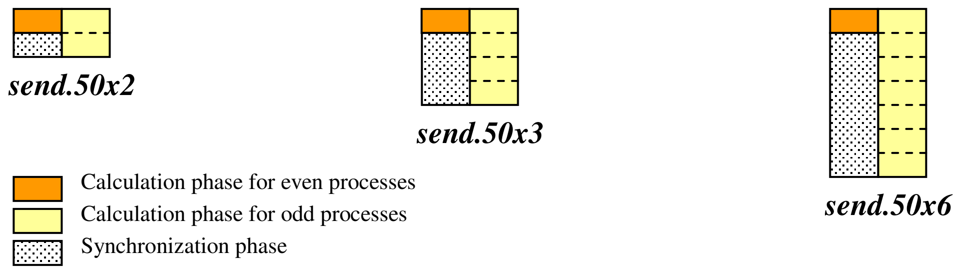

send.50x2
send.50x3
send.50x6
Calculation phase for even processes
Calculation phase for odd processes
Synchronization phase

Figure 3.10 Relation between the amount of time spent in the calculation phase for odd and even processes

Figure 3.10 shows the relation between the times spent in the calculation phase for the odd and the even processes. By varying the duration of the calculation phase of the odd processes it was generated different degrees of imbalance. In order to increase the imbalance degree, the calculation phase was increased by two, three or six times greater than the rest. The number 50 indicates the percentage of processes that are imbalanced. The percentages evaluated were 50 and 30 %.

3.6.1.2 Application classification

Depending on the scheduling policy evaluated, it is interesting to vary different aspects of the load and the applications. For this reason the applications are classified according to different criteria:

a) Communication degree

To establish the communication degree of the applications, the percentage of time spent in the execution of MPI operations as well as the type of the communications (collective or point-to-point) was measured.

In order to make this measurement each application with as many processors as processes was executed in isolation. Process migrations were not allowed. The results are shown inTable 3.1.

Table 3.1 Percentage of time spent in MPI operations

	% MPI	%Collective	%Point-to-Point	Comm. degree
ep.B.64	6%	6%	0%	Low
ft.A.64	30%	15%	15%	Medium
mg.B.64	38%	4%	34%	Medium
cg.B.64	40%	0%	40%	High
bt.A.64	46%	7%	39%	High
sweep3D.64	47%	9%	39%	High
lu.A.64	51%	0%	51%	High

b) Type of applications depending on their sequential execution time

The execution times of the applications executed in isolation with different number of processes are shown in Table 3.2. The executions were made with as many processes as processors. Applications were classified in long and short in according to their sequential execution time as can be seen in Table 3.3.

Table 3.2 Execution times in seconds for the applications varying the number of processes

	1	**8**	**9**	**16**	**25**	**32**	**36**	**49**	**64**
LU.A	1940	168		61		33			23
LU.W	160	20		12		11			6
MG.B	255	60		28		15			9
BT.A	2441		300	185	100		66	50	46
FT.A	89	28		10		7			3
EP.B	373	49		24		14			7
CG.B	4385	475		180		88			57
CG.A	85	7		3		2			2
Sweep3D	50	6		5		5			5

The applications were classified as ***short*** if their secuential execution time was less than 10 minutes and were classified as ***long*** if their sequential execution time was greater than 30 minutes.

Table 3.3 Application classification depending on their sequential execution time

	TYPE
LU.A	*LONG*
LU.W	*SHORT*
MG.B	*SHORT*
BT.A	*LONG*
FT.A	*SHORT*
EP.B	*SHORT*
CG.B	*LONG*
CG.A	*SHORT*
Sweep3D	*SHORT*

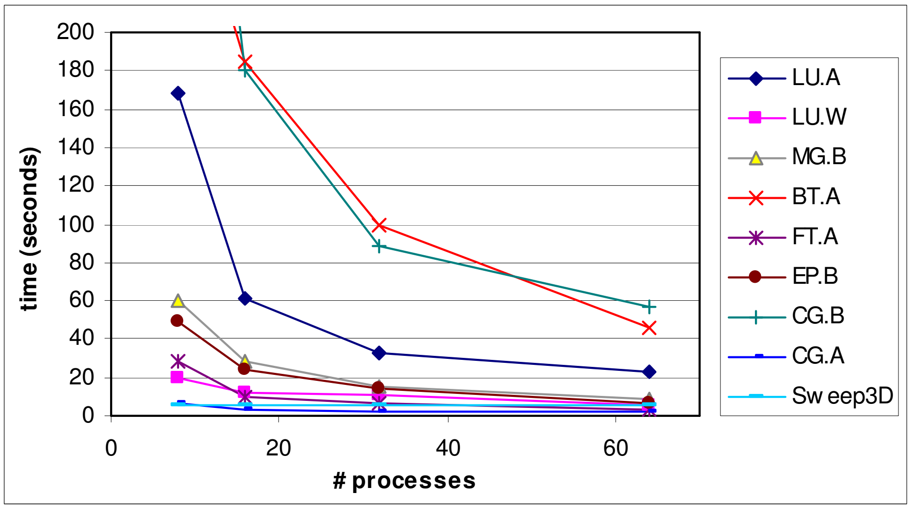


Figure 3.11 Scalability of the applications

Figure 3.11 shows the scalability of the applications analyzed and Figure 3.12 shows their speedup. It is important to notice that long applications have better speedup than short applications.

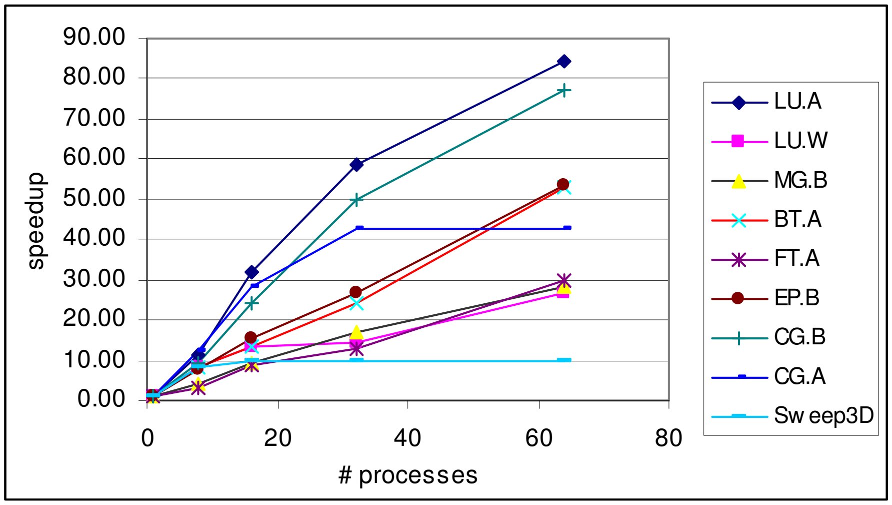


Figure 3.12 Speedup

The applications analyzed cover all the types of the classifications proposed, so they were considered enough for the evaluations of this work.

The number of processes used for each application varied depending on the evaluation. When evaluating rigid jobs the number for fixed to be the closest to the maximum in the available pool of processors (i.e. for a pool of 60 processors, the maximum for the cg is 32). When evaluating moldability, the applications were allowed to choose a range between 1 and the maximum not greater than the total number of processors.

3.6.2 Workloads design

The workloads used in this dissertation represent the execution of jobs with arrival times according to a Poisson distribution.

Equation 3.3 shows how the arrival times of jobs are calculated. P is the total number of processors in the system, being 64 at the most. The variable *U* is the machine load generated and it is indicated with a number between 0 and 1. The variable T_i^1 t represents the sequential execution time of application ***i***. The *Frac* constant is the percentage of the load that this application generates within the load and it is a number between 0 and 1.

$$\lambda_i \times \mathrm{T}_i^1 = \mathrm{P} \times U_i \Rightarrow \lambda_i = \frac{\mathrm{P} \times U_i}{\mathrm{T}_i^1} \Rightarrow \lambda_i = \frac{\mathrm{P} \times U_i \times Frac}{\mathrm{T}_i^1}$$

Equation 3.2 Calculation of the arrival times for the jobs of a workload

In order to generate the trace with the arrival times, it was implemented an application that receives as a parameter list of applications, a number $1/\lambda_i$ and the maximum desired duration for the workload expressed in seconds.

For example for a workload composed by four different applications: {cg.B, lu.W, bt.A, sweep3D}, where each one had 25% of the load generated by the workload (for a 60% of machine load), the first 90 seconds of the trace looks like this:

```
0 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
1 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
2 14 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1
3 17 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
4 33 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
5 36 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
6 38 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1
7 42 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
8 46 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
9 52 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
10 60 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 -1 -1 -1 -1
11 62 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
12 65 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
14 78 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
15 86 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
16 90 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
```

Figure 3.13 Example of a workload with the arrival times

The second column, as it was showed in Figure 3.3, are the arrival times of each application determined in the 14th column.

The workloads used in this work were designed with a maximum time between 600 and 1200 seconds. Nevertheless this upper limit is just to express the moment at which it is launched the last job of the workload. For this reason the workload can eventually finish later (i.e. the last job has an arrival time at the second 900, but could be actually executed at the second 2000 because of the unavailability of the resources). The *launcher* waits for the termination of the execution of the last job of the trace. None job of the workload is discarded for the evaluations.

3.7 Summary

In this chapter it was presented the implementation and the functionality of the elements that compose the execution environment of this work.

It was presented the queueing system used in this work (the *launcher*), a resource manager (*CPU Manager*) and the *VMruntime* library. It was described also the characteristics of the applications used for the evaluations and their classification according to the evaluation requirements. And finally it was made a detailed description of the mechanism followed for the generation of the traces used for executing the workloads.

The *launcher* is the queuing system at user level. It is in charge of applying the job scheduling policies, making the necessary decisions with respect to the number of processes, maximum multiprogramming level, when to execute the applications and the order of execution. The *launcher* takes all these decisions based on information obtained from the system and from the *CPUM*. Using the *launcher,* the job scheduling policies were evaluated in a controlled form and under the same conditions.

The *CPUM* is a resource manager at user level which is in charge of the processor allocation. The *CPUM* communicates with the *launcher* and the *VMruntime* library through shared memory.

The *VMruntime* library is used to get control of the applications by applying an interposition mechanism. In this way it is neither necessary to recompile applications nor to modify the message passing library. Through this runtime library the calls to the MPI library and the sginap routine are intercepted. The wrappers of these functions implement the processor scheduling policies and the process mapping.

The applications used in the evaluations were classified following several criteria based on characteristics like communication degree or sequential execution time. These classifications were used for the construction of the workloads, according to the objectives of the evaluations in each case.

Chapter 4 *VIRTUAL MALLEABILITY*

Abstract
This chapter presents the first contribution of this work, which consists of a mechanism named Virtual Malleability, that allows applications to adapt to the variations of the size of the assigned processor partition.
The mechanism is composed by two other techniques: 1) Self Coscheduling, which arises from the combination of space-sharing and coscheduling techniques; 2) Load Balancing Detector, which is in charge of deciding at runtime the balance degree of an application in order to organize processes in local or global queues.

4.1 Introduction

In order to obtain better use of the machine and synchronization between the processes of a parallel work, a typical scheduling strategy is to assign jobs to processor partitions for its exclusive use; these are named the *space-sharing* policies [LASM02]

If parallel jobs are rigid [FEIT97], the *space-sharing* policies assign static partitions to them. This leads to processor fragmentation [ZHFM00], which can be alleviated by applying backfilling techniques [SSKH02]. The backfilling techniques consist of forwarding jobs in the wait queue, when the job at the head of the queue cannot take advantage of the available processors, provided they will not delay this one.

Moldability [CIRN01] can also reduce fragmentation as the job sizes are adapted at the beginning of the execution to the available resources. Nevertheless, this facility has the disadvantage that once jobs start execution, they are assigned a number of processes, which cannot be changed during the execution even though the load of the system varies. On the other hand all the applications do not support moldability. Even more, this facility is not available in all the production systems.

Malleable jobs are the only that can adapt to load changes, eliminating fragmentation completely. Applying this to dynamic scheduling strategies helps to improve the response time of the system when the load varies.

The most popular programming models, for parallel jobs, used in high performance supercomputing centers are MPI [MPIF94] and OpenMP [OPENMP]. In [COML00] they apply successfully malleability to OpenMP applications. This cannot be possible with MPI jobs, since they are moldable at the most.

In the case of OpenMP, malleability is offered by the runtime library and hardware support. The redistribution and the access to data are made in a transparent form for the programmer, by applying cache coherence mechanisms of the underlying architecture.

In the case of MPI it is more complex since the data must be distributed between the processes in an explicit form. Whenever the number of processes is modified at runtime, it is necessary that an explicit redistribution of data is programmed. This redistribution of data requires a deep knowledge of each application by the programmer.

Virtual malleability is a mechanism by which a job is assigned a processor partition dynamically, where the number of processes can be greater than the number of processors. The partition size can be modified during execution of the job according to the external requirements of the system, such as the load, by means of the variation of the multiprogramming level (MPL).

The mechanism is composed by two techniques, one that allow applications vary dynamically its processor partition (*Self Coscheduling*) and an algorithm that decides at runtime the process organization depending on the balance degree of the application (*Load Balancing Detector*).

Figure 4.1 shows the execution of two jobs under self coscheduling with processes organized in local queues. As soon as *job B* finished its execution, its processors were freed, so *job A* was able to expand and use the newly available processors. It can be seen that each job competes with itself for the use of the resources.

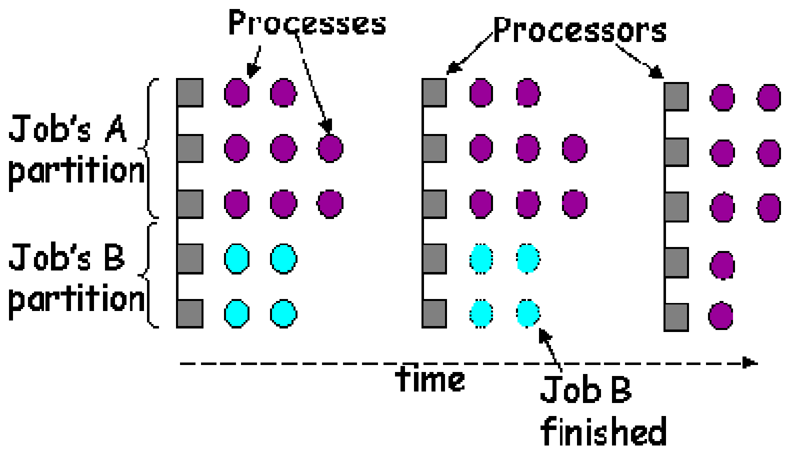


Figure 4.1 Example of processor scheduling applying self coscheduling

As the partition size can be smaller than the number of processes, proceses must be organized in queues. The alternatives analyzed were organizing the processes in local queues per processors in the assigned partition or organizing the processes in a global queue per application.

It is well known that the performance of one or the other alternative queue types depends on the balance degree of the application [GUTU91]. In this work, it was developed a mechanism named the *Load Balancing Detector* (*LBD*) [UtCL0905], which dynamically measures the balance degree of a job. If the job shows an imbalanced behaviour a global queue is applied, otherwise local queues per processor are applied.

The *virtual malleability* mechanism was implemented and evaluated under different workloads varying the multiprogramming level, the machine utilization, and the communication degree of the job. It was also compared to other alternatives of the bibliography. The results showed that a parallel job with a high degree of communication obtained better performance when competing with itself for the use of resources than with other jobs (*coscheduling techniques* [FEJE97], [DUCM98], [DUAC96], [SOPC98]). On the other hand, the dynamic mechanism of decision of the queue type, obtained some benefit when applied to regular applications compared to using a fixed approach (local or global queues).

The proposal of this chapter is evaluated from the point of view of the application, so a simple job scheduler was enough, with a *FCFS* policy and where the partition size assigned to each application was equal to the number of processes divided into the multiprogramming level (MPL). That is to say, each application is always executed with the minimum partition based on the MPL and the number of processes.

4.2 Related work

This section describes the state-of-the-art related to *virtual malleability*. The related work discussion is divided into processor sharing strategies that correspond to the proposal of *self coscheduling* [UtCL0904] , load balancing strategies and classification of jobs at runtime according to their balance degree that correspond to the *LBD* proposal.

4.2.1 Processor sharing policies

The main strategies existing in the literature for processor sharing can be classified in three groups: *time-sharing, space-sharing* and *coscheduling* o *gang scheduling.*

When there are more processes than processors, the *time-sharing* algorithms multiplex in time the use of processors between the jobs. This strategy obtains good performance for sequential jobs since it reduces their response time. Nevertheless, for parallel jobs the performance degrades due to the lack of synchronization and the number of process context swiching frequency when sharing processor.

The *space-sharing* techniques reduce the context switching effect by partitioning the set of available processors between the different jobs that are in the system. The partitioning can be static or dynamic.

When the partitioning is static a job is assigned a set of processors for all the execution, whereas in a dynamic one, the processors can be reassigned depending on the job requirements and the system conditions. Dynamic processor allocation policies have demonstrated to have good performance especially on malleable jobs, such as OpenMP, where the number of processes of a job can be adapted to a variable number of processors. But in the case of MPI, the job must wait for enough resources or it must reduce its parallelism. This last option is only possible if the job is malleable, which cannot be the case for MPI jobs.

As always the best option arises from the combination of the existing approaches, in this case between the *time-sharing* and *space-sharing* processor sharing policies. These are the *coscheduling* and *gang scheduling* processor sharing policies. In *gang scheduling* all the communicating processes are executed simultaneously, the processors are shared in time between the different jobs. *Coscheduling* is its relaxed version, whose strategy is to try to maintain scheduled at the same time as much as communicating processes as possible without an explicit synchronization. The coscheduling algorithms can be classified as: explicit scheduling, local scheduling and implicit or dynamic coscheduling.

Explicit scheduling was proposed by by Feitelson and Jette in [FEJE97], it coordinates the scheduling of communicating processes through a static global list with the order of execution of the jobs and simultaneous context switching in the processors. The list of communicating processes is necessary to know in advance, before the execution of the job.

The global synchronization is essential not only for fin grain parallel jobs, but also for those coarse grain parallel jobs. A centralized scheduler is a complicated approach mainly in distributed systems.

The local scheduling appears in [GUTU91] and was applied to distributed systems, where each node has its own operating system. Each scheduler takes local scheduling decisions in an independent form. The performance of fine grain parallel jobs is degraded as there is no mechanism at all of synchronization between processes in different nodes.

Implicit or dynamic coscheduling is an intermediate approach developed in UC Berkerley and the MIT [DUCM98], [DUAC96], [SOPC98]. The scheduling decisions are taken based on local events of communication. The implicit information available locally is related to the arrival of messages and the round trip time of a message in the network. The synchronization is guided through dynamic coscheduling without having any explicit synchronization between the processes. This mechanism can be applied to SMPs as well as to clusters.

Table 4.1 Action combination based on messages events

	Event: Waiting for a message		
Event: Message arrival	***Busy Wait***	***Spin-Block***	***Spin Yield***
No explicit Re-Schedule	Local [GuTU91]	Spin Block [ArCu01]	Spin Yield [ZSMF00]
Interrupt & Re-Schedule	Dynamic Coscheduling [SoPC98]	Dynamic Coscheduling - Spin Block [NBSD99]	Dynamic Coscheduling - Spin Yield [NBSD99]
Periodically Re-Schedule	Periodic Boost [NBSD99]	Periodic Boost - Spin Block [NBSD99]	Periodic Boost – SpinYield [NBSD99]

As can be observed in Table 4.1, presented in [NBSD99][ZSMF00], the *coscheduling* techniques result from the combinations of two components in the interaction between scheduler and communication mechanism: what to do when waiting for a message and what to do on message arrival.

When waiting for a message, one possible alternative is doing *busy waiting*, that is polling for the message forever. Another alternative is to poll for the message during an interval and after that if the message hasn't arrived, the process blocks. This is named *spin blocking* and it was proposed in [DUCM98], [ArCu01]. They implement this strategy using a message-passing programming model, which provides them an instant mechanism of knowing if the target is currently running or not. The spin time is chosen in such a way to optimise performance. They show that this scheme can work well with bursty communication jobs. Once a process is blocked, the scheduler can give tips on which process to run next as in the s*pin yield* [NBSD99].

On message arrival, an alternative is just to ignore the message or do re-scheduling. It can be based on interruptions by preempting the currently running process and giving the processor to the target process of the message or based on priorities where some monitor process manipulates the priorities depending on the unconsumed message queue of each process as in the *Periodic Boost* [NBSD99][ZSMF00]. In order to eliminate the interruption overhead generated by the message arriving they propose an entity kernel that periodically examines the queue of unconsumed messages of each process and raises process priorities based on some heuristics, for example the first process in the queue with unconsumed.

The evaluations made in [ZSMF00] were done using a small number of processes and processors, from 8 to 16 and the workloads used involves bursty and medium fine grain jobs. The platforms evaluated are networks of workstations where latency has to be considered and process migration may result of high cost. On the other hand the workloads were composed by at most four jobs arriving at the same time.

4.2.2 Load balancing and job classification according to their balance degree

Local queues per processor are the natural alternative for machines with distributed memory. Even more, it is also possible to apply to machines with shared memory, given the existence of certain local memory. In this scheme, each processor has exclusive use of its local process queue, eliminating queue contention and necessity of locks. Nevertheless, decisions about process mapping and processor sharing have to be made in order to satisfy the communication requirements and to minimize synchronization overhead.

A very extensive discussion exists about the use of global and local queues in [FEIT97]. Local queues were used in Chrysalis [LESB88] and Psyche [SLMB90] in the BBN Butterfly. The key to apply local queues is the load balancing between processes of an application. The load balancing depends on the balance degree of the job and therefore process mapping to processors. In [BKSH01] they balance the load by the creation of threads at loop level.

A global queue is something simple to implement in shared memory machines, but it is not the case for distributed memory machines. The main advantage of using global queues is that they provide automatic load balancing or load sharing as they call in [FEIT97]. However, this approach suffers from queue contention, lack of memory locality and possible locks overhead.

In [SQNE91] they show that to ignore the affinity can result in significant performance degradation. This point also is discussed in [VAZA91]. Nevertheless, this point is not crucial, since in local queues, the cache has been emptied by a number of other processes of the local queue that have executed before [GUTU91].

In [BLAC90] they describe a particular implementation of a global queue in the context of the Mach operating system. They use a global queue based on priorities according to processor utilization and system load.

A combined approach appears in [BRCR91], where in each context switch, a processor can choose the next process to execute from a local queue or from a global one depending on a priority system.

In [FFPF03] they classify jobs at runtime depending on their communication degree. In this way they generate sets of processes ready to be scheduled at the same time following a *gang scheduling* strategy.

4.3 Virtual malleability

This section is dedicated to the description of the proposal of this chapter: virtual malleability. This mechanism is composed by: *self coscheduling* [UtCL0904], which is involved in doing processor scheduling and *LBD* [UtCL0905], the mechanism that dynamically decides the process queue type.

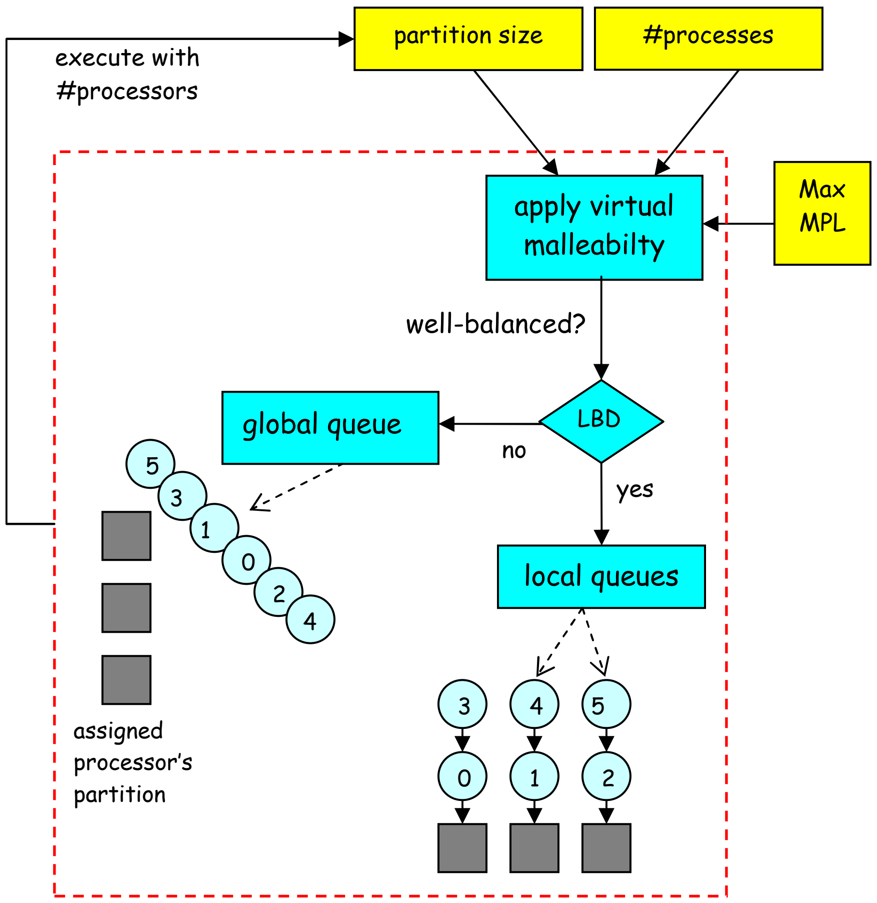


Figure 4.2 Virtual malleability mechanism and its parameters

Figure 4.2 shows graphically how the virtual malleability works. The mechanism receives as parameters a partition size, the maximum MPL and the number of processes.

As soon as the job begins execution, the *LBD* mechanism determines dynamically if it has a well-balanced or imbalanced behaviour thus applying the corresponding queue type.

Once the type of queue is established, the processors are shared between the processes of an application using the self coscheduling mechanism. Which consist on making the processes compete with themselves for the use of resources by applying the coscheduling techniques.

4.3.1 Coscheduling heuristics

As the number of processors in a system can be smaller than the total number of processes, it is required a policy that determines how processors are shared between the processes of an application. Traditionally applications shared resources between them by time-sharing processors and synchronizing using coschduling approaches. Another possibility is to assign partitions to each application for its exclusive use. In this way the resources are shared just between processes belonging to the same application and consequently they will have to compete with themselves for its use.

Coscheduling decisions:

The coscheduling techniques make scheduling decisions based on local events generated by the arrival or sending of messages. These decisions are concerned with: a) how to wait for a message; b) when to free the processor; and c) how to choose the next process to run.

a) How to wait for a message:

   Spin blocking: Do busy waiting during a spin time. After the spin time expires, if the message hasn't arrived, the process blocks.

   Blocking immediately: When executing the receive operationg, if the message is not available, the process blocks immediately.

b) When to yield the processor:

   Time-slice: The process executes during a time-quantum, after that it frees the processor.

   Event-guided: There isn't any time limit when executing a process. The process ejecutes until it arrives to a blocking operation like a message receiving. From this point it depends on what it does to wait for a message.

c) How to select the next process to run:

   Round-robin: The next process in the queue. This is equivalent to the one that has executed less recently.

   Unconsumed messages: The process that has the greatest number of unconsumed messages.

Sender: The process that has to send the message by which the currently running process has blocked. In case the sender process is also blocked, then is chosen the next process in the queue with unconsumed messages.

Affinity: The process that has last executed on that processor. This only has sense when applying global queues. In case this process is also blocked, then is chosen the next process in the queue with unconsumed messages is selected.

4.3.2 Self coscheduling

This mechanism exploits the process scheduling at processor level with the aim of minizing the loss of performance when the total number of processes is the system is greater than the number of processors.

Self coscheduling arises from the combination of *coscheduling* techniques [DUCM98], [FEJE97], [NBSD99] with *space-sharing* policies [GUTU91] in a dynamic environment, where the size of the assigned partition to an application can vary at runtime.

The coscheduling techniques as it was mentioned in the related work are based on scheduling simultaneously the greatest possible number of communicating processes without any explicit synchronization. The scheduling decisions are based on local events such as how to wait for a message, what to do when the message arrives, and which process is selected to execute next. The processor is time shared between the processes of the different applications.

Table 4.2 Comparison of the characteristics of coscheduling and self coscheduling techniques

	processor sharing	# processors	process organization
coscheduling	time	# processes	local queues
self coscheduling	space	# processes / MPL	local / global queues

Table 4.2 shows the comparison of the main characteristics of coscheduling and self coscheduling approaches. The number of processors assigned to an application under coscheduling is equal to its number of processes, while under self coscheduling it depends on the the number of processes and the MPL. The processes under coscheduling approaches are organized in local queues at each processor; while under the self coscheduling approach, the processes can be organized either in local queues per processor or in global queues per application. Another difference is that when processes are organized in local queues, they all belong to different applications if coscheduling approaches are applied, while under self coscheduling they all belong to the same application.

4.3.3 Load balancing detector (LBD)

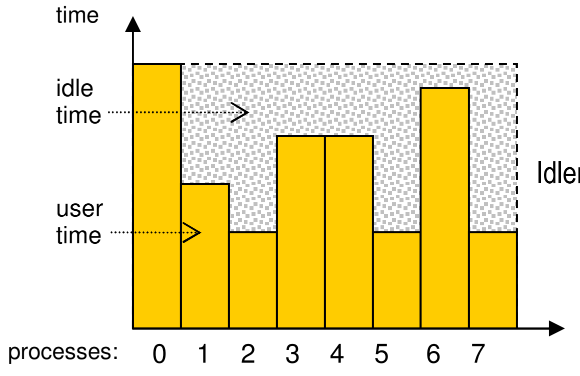
This section describes two alternatives of organizing the processes and an algorithm that decides dynamically, given an application, the best alternative of those two.

The two alternatives analyzed to organize processes were: local queues per processor or global queues per application. It is well known that the performance of each approach depends mostly on the load balancing degree of each application. For this reason it have been developed a mechanism which measures it dynamically and depending on the result, applies the appropriate queue type.

When a parallel application begins execution, the processes are created and the data is distributed. During this period the application has an irregular behavior, in the sense that their processes don't register an identical amount of work, the processors are not equally loaded. This phase of the execution of an application is said to be **chaotic**.

Therefore any measure on the balance degree at the initialization phase of an application is not representative of the application as it would show always an imbalanced behaviour. So a mechanism to differentiate the initial phase of the execution from the regular execution phase was necessary to design. For the construction of such mechanism, just applications with regular behaviour were considered. This means that once the applications finish the execution of their initialization phase, they have a regular behavior, well-balanced or imbalanced, during the rest of its execution. In this work, applications that show both behaviours during different execution phases were not considered.

During the initialization phase the use of the resources is irregular, thus generating an irregular number of context switches between processes. Nevertheless, once this phase finishes, the number of context switches becomes almost constant, and consequently the variation is constant. For this reason, the measurement of the variation of the number of context switches was used to detect the moment the application enters in a regular computational phase. This was named in this work, the coefficient of variation of context switches (*CVCS*). Once the initialization phase is finished, the load balance degree measurement of the application is considered to be representative of the rest of the execution.



$$\text{Idleness \%} = \frac{\sum_i user\_time_i}{\max(user\_time_i) \times \# processes}$$

Figure 4.3 Idle and user time of an application (left) and *Idleness %* equation (right)

The user time of a process reflects the time that a process has spent doing useful work. Figure 4.3 shows graphically the execution of an application with eight processes. The bars represent the time spent by the processes doing useful work, the rest of the area represent the time the processes were waiting for the process 0, which had the longest execution time. During this time, the processors assigned to processes 1 to 7 were idle. In Figure 4.3 the equation for the calculation of the percentage of idleness is also shown. This percentage is the sum of the times each process was waiting for the termination of the process with the longest execution time. This percentage gives an idea of how much time an application was wasting resources, because of imbalance. This was named in this work *idleness percentage* (*IP*).

Each specific *IP* for each application used in the evaluations of this work, was previously calculated. So using this knowledge, an *IP threshold* was deduced empirically to classify the applications in well-balanced and imbalanced.

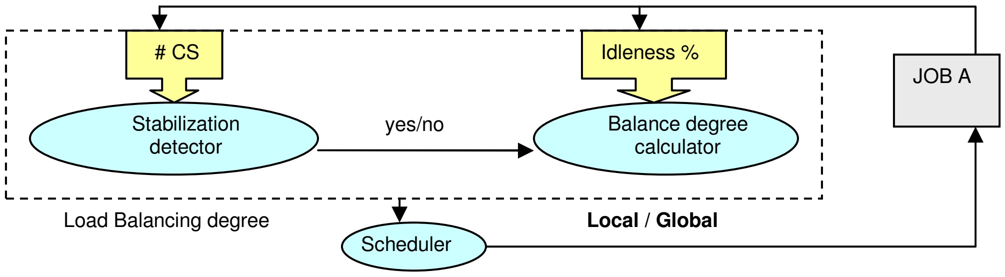


Figure 4.4 CPUM internal structure with the LBD mechanism

Figure 4.4 shows the internal structure of the *CPUM* with the *LBD* mechanism. The *stabilization detector* makes periodically the calculation of the *CVCS* at runtime using the information provided by the runtime library with the number of context switches. Once the stabilization detector decides that the initialization phase has finished, it informs about that to the *Balance degree calculator*. From this moment, the *Balance degree calculator* is allowed to calculate the *IP*.

The *LBD* compares the calculated *IP* with the *IP threshold* to classify the job as well-balanced or imbalanced. After that if the job was classified as well-balanced the *LBD* applies local queues per processor; otherwise it continues applying a global queue to it.

4.4 Implementation characteristics

In this section are described the main characteristics of the implementation of the mechanism of virtual malleability, its components and the relation between them. There is a section dedicated to the description of the coefficients calculations (*CVCS* and *IP*). And also a section dedicated to the comparison of different heuristics to select the next process to execute in the local and global process queue.

4.4.1 Relation between the components

In Figure 4.4 can be seen scheme with the components that take part in the construction of the mechanism of *virtual malleability*.

The queueing system or *launcher* selects the jobs from the wait queue for execution. The selected job enters immediately under the control of the *CPUM* and forks the number of processes indicated by the *launcher.* The *CPUM* assigns a processor partition to the job with a minimum size calculated following the Equation 3.1. The *CPUM* applies also the *LBD* mechanism. The runtime library is in charge of doing the process mapping and the processor scheduling.

In this chapter were considered just rigid jobs. As it was commented before the objective was to evaluate the mechanism of *virtual malleability* from the point of view of the application. For this reason a very simple job scheduling policy, the *FCFS*, was applied. The jobs are executed as soon as possible and in the same order as they arrive applying the maximum *MPL* allowed in the system.

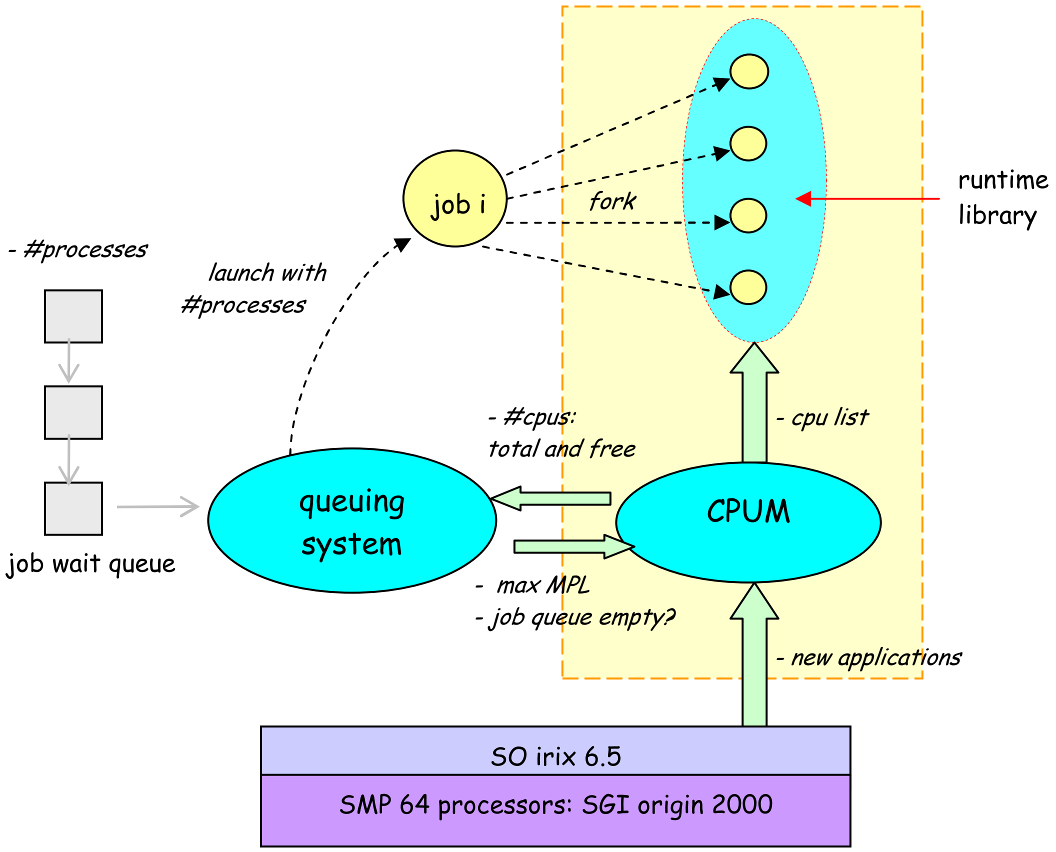


Figure 4.5 Virtual malleability in the execution environment

Given the *MPL* and the number of processes, the minimum partition size is completely determined.

In order to evaluate the *virtual malleability* mechanism and to compare it with other alternatives, the applications are not allowed to expand if there are jobs in the wait queue. This is valid even though there were free processors. This situation can occur, when the number of free processors doesn't satisfy the minimum partition size of the first job in the queue. In this way the overhead generated when applications expand and/or shrink is eliminated and the evaluations are centered just when the applications are shrunk.

The resource management is done by the *CPUM*. It informs the *launcher* about the number of free processors and it is informed about the state of the wait queue, to decide if an application is able to expand.

In order to control the application an interposition library (*DiTools* [SENC00]) is used. This library allows the applications to intercept functions of the MPI message passing library, as well as the *sginap* routine. The VMruntime library implements the wrappers of all the intercepted functions.

The system routine *sginap*, is invoked by the *MPI* library whenever it executes a MPI blocking operation, such as waiting for a message that has not arrived yet. The context switching is made through the wrapper of this function by blocking the currently running process. As the implementation is done outside the operating system, the processs are blocked and unblocked manually.

The sginap wrapper is also in charge of counting the number of context switches and the user time for each process. These numbers are used for the calculation of the *CVCS* and the *IP* respectively. Figure 4.6 shows the code for the *sginap_wrapper* routine. This routine inhibits the execution of the original *sginap* routine.

```
sginap_wrapper

    sginap_calls++;
    now = current_time();
    if (spin_time>0 && (now-ini_wait_time) < spin_time)
            return;
    end if
    if (local_queues) then
            set_lock (mycpu);
            pid_next = choose_local_next (policy, mycpu.queue);
            set_unlock (mycpu);
    else if (global_queue) then
            set_lock (myappl.queue);
            pid_next = choose_global_next (policy,myappl.queue,mycpu);
            set_unlock (myappl.queue);
    end if
    if (pid_next != me)  then
            unblock (pid_next);
            block (me);
    end if

fin sginap_wrapper
```

Figure 4.6 Sginap wrapper routine code

As the *sginap_wrapper* routine is used to do the context switching, it also selects the process to run next. According to the load balance degree of the application, the next process is selected from a local queue per processor or from a global per application. It is important to notice that the locks generate less overhead in the local queue than in the global queue approach.

About the manual context switching there exist certain overhead, since it is performed through system functions. Moreover, the two processes overlap execution for a moment. Nevertheless, the process that yield the processor is just performing busy waiting, so it does not make any useful work.

Self coscheduling applies blocking immediately, so busy waiting is not applied. Anyway, alternatives with spin times greater than zero were implemented and evaluated as well.

As can be seen in Figure 4.6 the *sginap_wrapper* routine checks the wait time, and if it is greater than a previously set spin time, then it blocks immediately the current process. The starting time is set in the *init_wait_time* variable. This value was initialized in the routines that wrap the blocking MPI functions, like the *mpi_recv.* Figure 4.8 shows the wrapper for the *mpi_recv.* A call to the original MPI function is done at the end of the wrapper. This is not the case for the *sginap* routine, where the call to the original function is never done.

```
int MPI_send_wrapper (void *buf, int count, MPI_Datatype datatype,
                                        int *dest, int tag, MPI_Comm comm)
{
      int ret;
      sub_unsent_msgs(me);
      ret=mpi_send_(buf, count, datatype, dest, tag, comm);
      add_unconsumed_msgs (dest);
      return ret;
}
```

Figure 4.7 Wrapper routine for the mpi_send

The *MPI* wrapper functions are used to count the number of unconsumed messages. These functions also count the number of *unsent messages* for each process. The *unsent messages* are the ones which have been already requested by other processes but they haven't been sent yet by this one. The information about which process has executed last in each processor is also kept. This is useful only when applying the global queue approach in order to implement the affinity heuristic.

```
int MPI_recv_wrapper (void *buf, int count, MPI_Datatype datatype,
            int *source, int tag, MPI_Comm comm, MPI_Status status)
{
    int ret;
    init_wait_time = current_time();
    add_unsent_msgs(source);
    ret=mpi_recv_ (buf, count, datatype, source, tag, comm, status);
    sub_unconsumed_msgs (me);
    sub_unsent_msgs(source);
    return ret;
}
```

Figure 4.8 Wrapper routine for the mpi_recv

The different heuristics to decide which process execute next are constructed by using all this information.

Each MPI process has an internal identifier which is named *mpi_rank*. This number is especially important when applying the local queues approach, since it is used by process mapping algorithm. The mapping is done in a round-robin fashion, that is to say, the process with the lowest *mpi_rank* is assigned to the first processor of the partition and so on. For a more detailed description about the process mapping algorithm used in this work, refer to chapter 3.

4.4.2 Evaluation of heuristics and performance comparison between global and local queues

In this section are described the evaluations of the heuristics listed in 4.3.1 to select the next process to run from a global queue. In addition it is presented a performance comparison when applying local and global queues to applications.

The objectives were, on one hand evaluate the performance of both queue approaches when applied to well-balanced and imbalanced applications. And on the other, obtain the best configuration for the *self coscheduling* when using global queues.

For the experiments of this chapter, the local queue approach was configured with *round robin* for selecting the next to run and *blocking immediately* for message waiting. This configuration was the one that worked best for well-balanced jobs as it is shown in the next section.

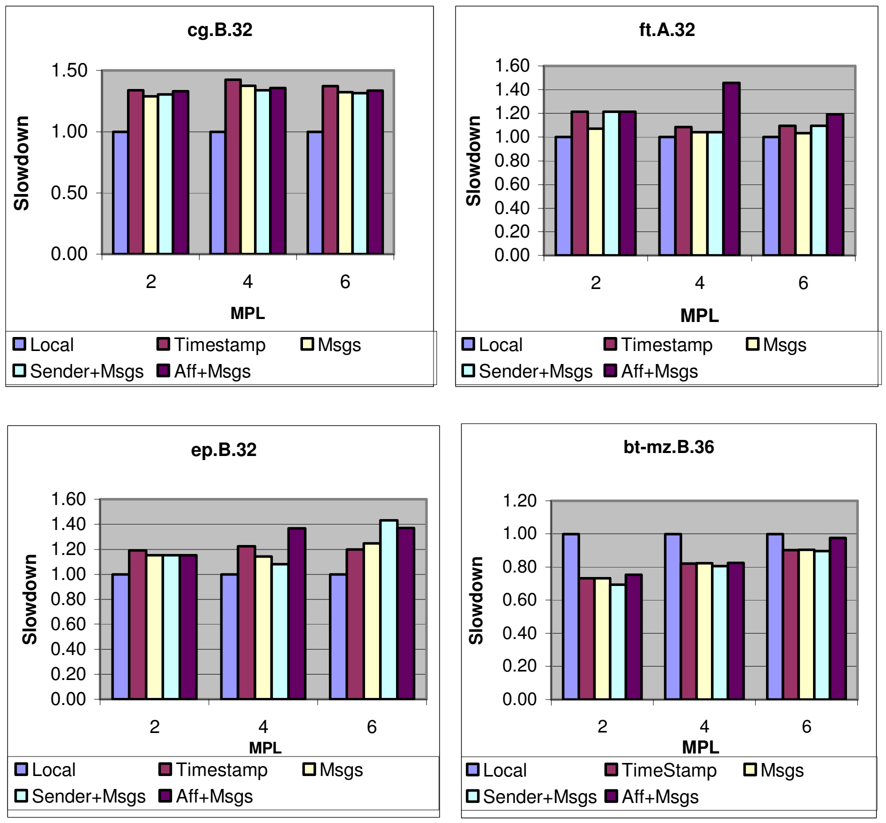


Figure 4.9 Performance comparison for different heuristics using global queues

The results of the evaluations for individual applications are shown in the graphs of Figure 4.9. On the x axis are the *MPL*s and on the y axis are the execution times normalized by the execution time on local queues. Each application was run in isolation using local and global queues with different heuristics. In addition they were evaluated using three different *MPLs*: 2, 4 and 6.

The first conclusion that can be taken is that the performance for well-balanced applications is better when running using the local queue approaches, while the performance for imbalanced applications (i.e. *bt-mz*) is better when running using the global queue approaches.

About the heuristics, it can be seen that *timestamp* (or *round-robin*) obtained the worst performance. This is because this heuristic does not take into account any process syncronization.

The best performance was obtained with the *sender* heuristic. It has the smallest number of context switches when compared with the rest of heuristics. The *sender* heuristic has the property of incrementing the probability of selecting the process that will enable a better synchronization between the processes of the parallel job.

Although global queues under the *sender* heuristic reduce the number of context switches, the locality provided by the local queues approach is not compensated when using well-balanced applications.

For the case of the *ft* application, even it is well-balanced, the performance obtained with both queue approaches was very similar. This application is composed mostly by global synchronizations, so all the processes need to synchronize frequently at the same time. Although, local approaches favour locality, the global approaches favour an equal distribution of the cpu time between the processes.

In the *ep* case, the differences between the two approaches were very small. This application performs calculation most of the time, so it does not depend on any synchronization. The number of context switches and process migrations is reduced.

Finally, about the MPLs, the best performance was obtained when it was set to 4.

4.4.3 Coeficient of variation of context switches (CVCS)

In this section the calculation of the *CVCS* coefficient is described.

The coefficient of variation of context switches or *CVCS* is used to detect the moment in which the application starts executing regular work. This means that the application has finished executing the initialization phase.

The applications are classified without any previous knowledge of them, at runtime and within the assigned partition, even though this is smaller than the number of processes.

As the calculations have to be done at runtime, and to ensure correct measurements, it is necessary that all the processes had a fair distribution of cpu time.

Guaranteeing equal access to all the processes of an application to the assigned processors is simple when having as many processes as processors. But if the partition is smaller, one possibility is to expand the application, make the calculations and then shrink it again. However, this is unacceptable due to the overhead introduced. On one side there is loss of affinity when expanding and shrinking the application and on the other, it would be necessary to force other applications currently executing to suspend or to shrink, generating even more overhead and lost of affinity. Moreover, this kind of movements would be necessary every time an application arrives to the system.

A global queue per application is applied at the beginning of the execution to ensure equal cpu time between the processes.

The *VMruntime* library counts the number of context switches from every process of each application and the *CPUM* calculates the *CVCS* as shown in Equation 4.1. This calculation is made every 10 milliseconds (every time the *CPUM* wakes up). As soon as this coefficient becomes constant, it is said that the application has stabilized and the *IP* is representative of the balance degree of the job.

$$CVCS = \frac{StdDev(\#ContextSwitches_i)}{Average(\#ContextSwitches_i)}$$

Equation 4.1 Calculation of the CVCS for each application

The graphs in Figure 4.10 and Figure 4.11 show examples of the *CVCS* for executions of well-balanced applications and imbalanced ones.

When calculating the *CVCS* it was applied always a maximum *MPL* equal to four. The heuristic selected to choose the next process to run from the global queue was *sender*, described in section 4.3.1.2. This heuristic has demonstrated to have the best performance as it can be seen in the performance evaluations of this chapter.

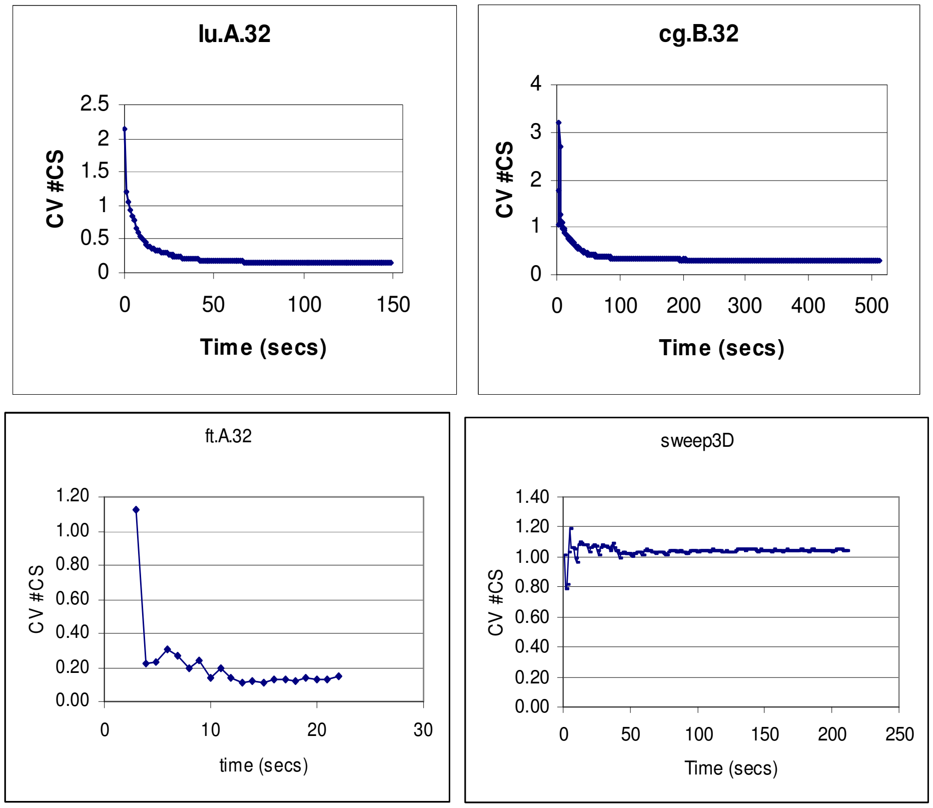


Figure 4.10 CVCS for well-balanced applications

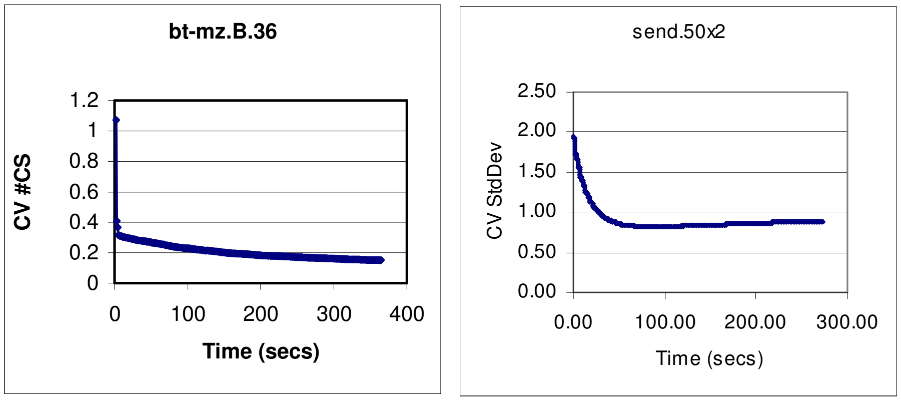


Figure 4.11 CVCS for imbalanced applications

Although the *CVCS* calculation reflects the initial state of the application as well as the moment that it begins with its regular execution, it does not say anything about the load balance degree of it. This is because the number of context switches depends on the frequency whereupon the processes execute blocking functions like waiting for a message or global synchronizations.

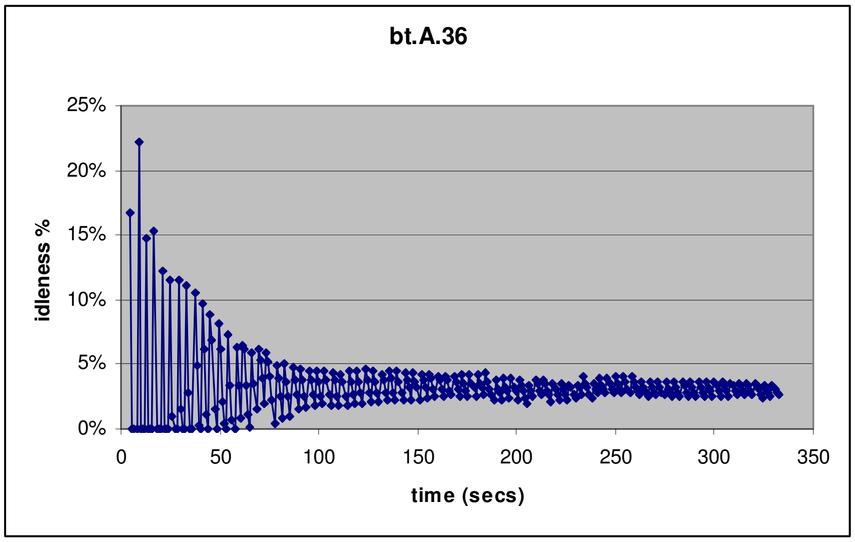
The *CVCS* is interesting for the mechanism because once the application starts doing regular work, it remains constant.

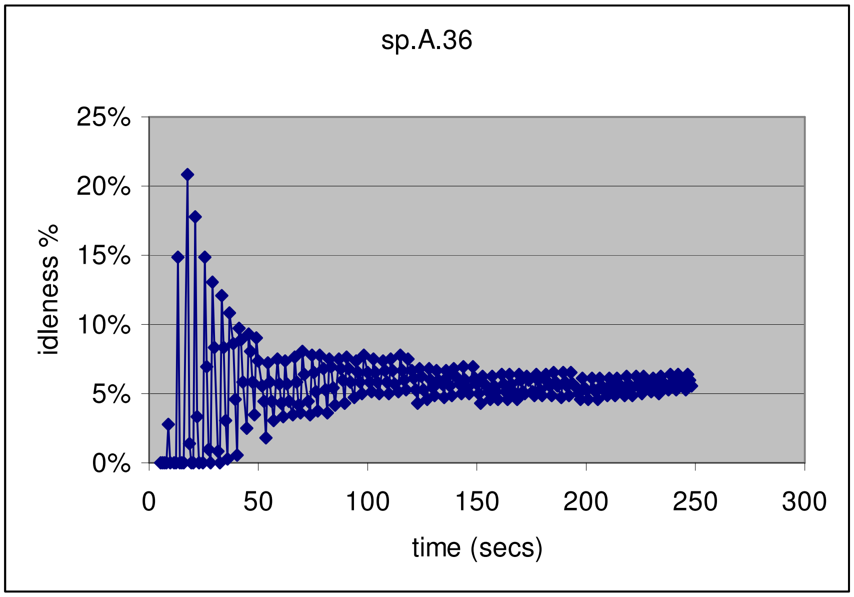
4.4.4 Idleness percentage (IP)

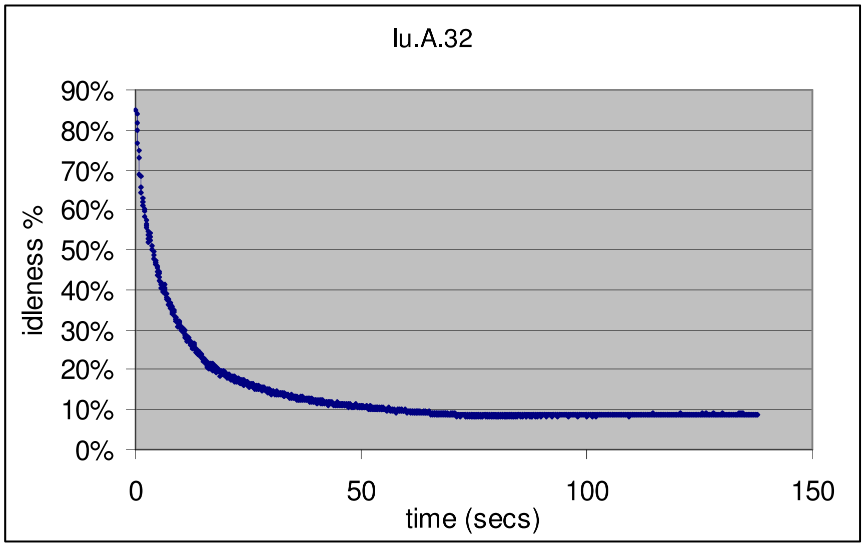
Once the application has finished its initialization phase, the *CVCS* becomes constant, so the *idleness percentage* (*IP*) becomes representative for the balance degree of the application.

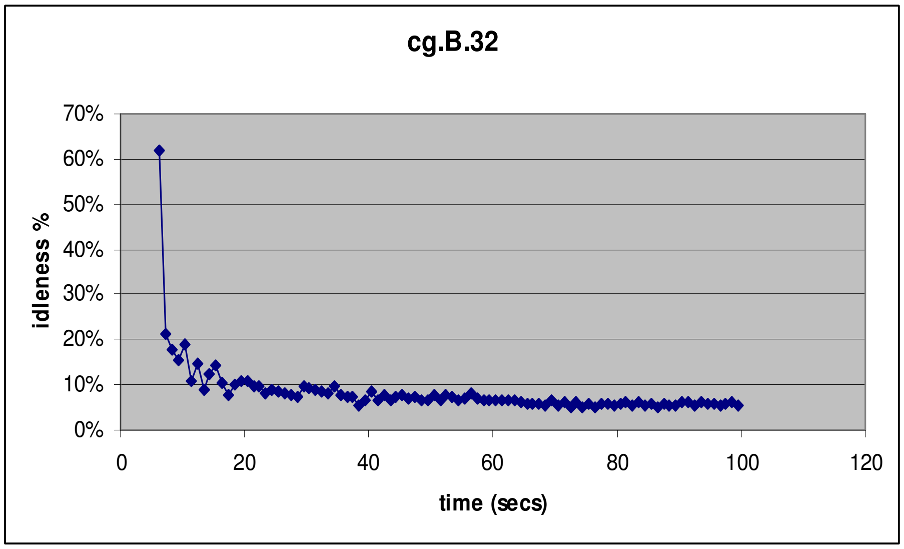
For this dissertation, just regular applications were considered. This means that the applications used have the same behaviour during all the execution. If one demonstrated that was well-balanced then it would be till the end, the same applied if it demonstrated to be imbalanced.

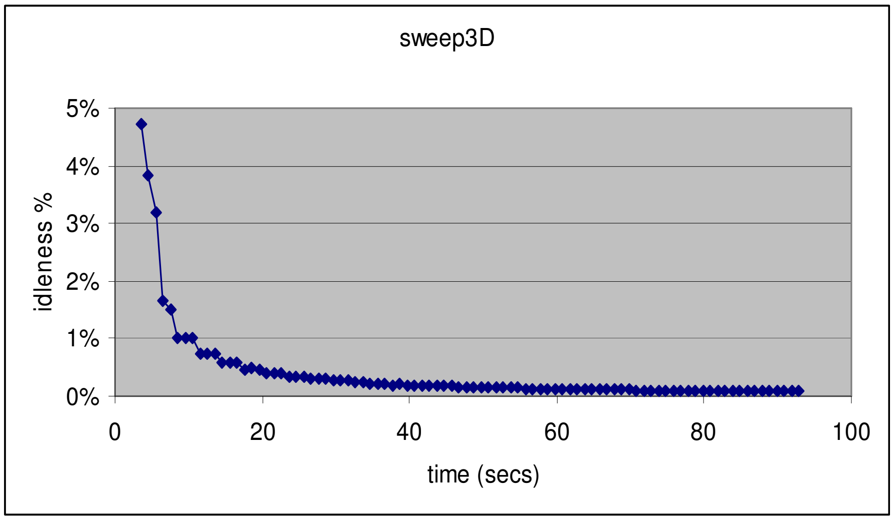
However, it would be possible to extend the mechanism for applications that have phases with different load balacing behaviour. In that case it would be necessary to monitor the *IP* during all the execution and switch between both queue types depending on the results. But, it would be also necessary to evaluate if the gain obtained when using the appropriate queue, surpasses the overhead generated when switching between both queue types.




sp.A.36
idleness %
25%
20%
15%
10%
5%
0%
0
50
100
150
200
250
300
time (secs)


lu.A.32
idleness %
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0
50
100
150
time (secs)


cg.B.32
idleness %
70%
60%
50%
40%
30%
20%
10%
0%
0
20
40
60
80
100
120
time (secs)



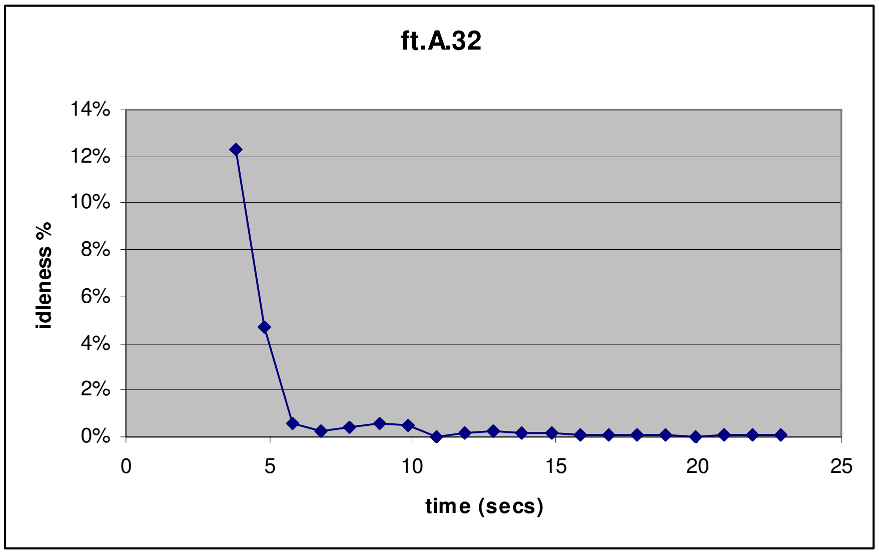
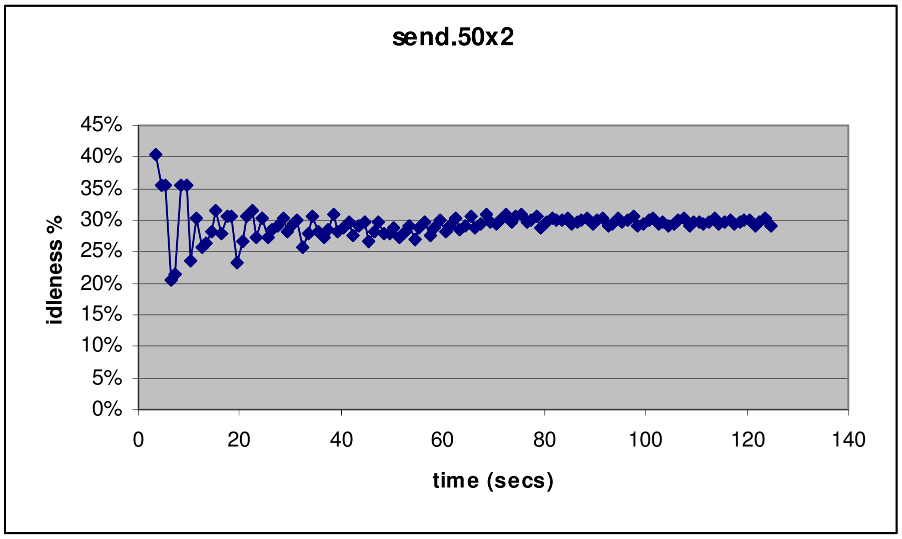


Figure 4.12 Evolution of the IP for well-balanced applications



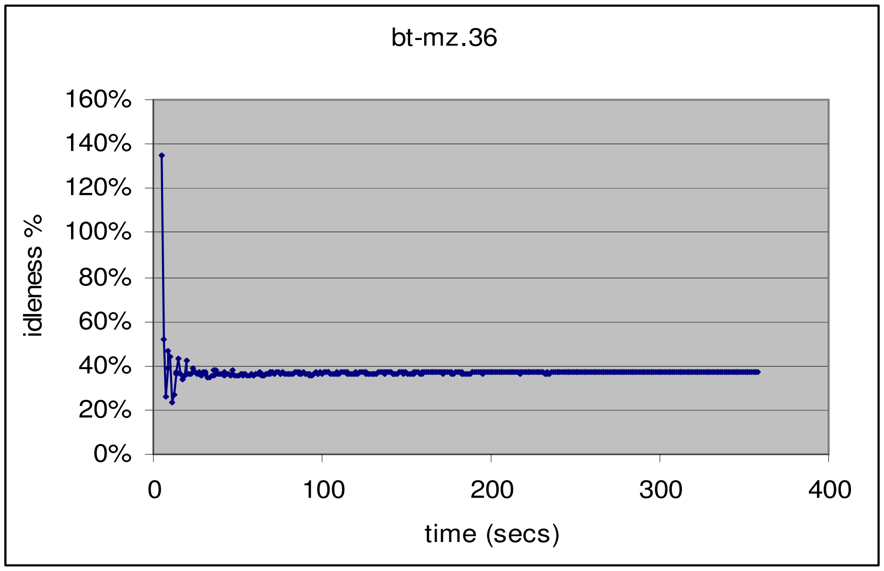


Figure 4.13 Evolution of the IP for imbalanced applications

Figure 4.12 and Figure 4.13 show the graphs for the *IP*s evolution for the executions of well-balanced and imbalanced applications. As can be seen, it is necessary to discard the *IP* corresponding to the beginning of the executions, as these would lead to incorrect classifications. Once the application has stabilized the *IP*, it remains constant until the end of the execution. The *IP threshold* was chosen empirically by measuring all the applications that are used in this work. The *IP threshold* is ***10%.***

The applications that obtain an *IP* greater than **10%**, are classified as imbalanced, and continue executing using a global queue approach. On the contrary, the applications that obtain an *IP* smaller than 10%, are reorganized in local queues, by applying the process mapping algorithm described in the previous section, which is shown in Figure 4.14.

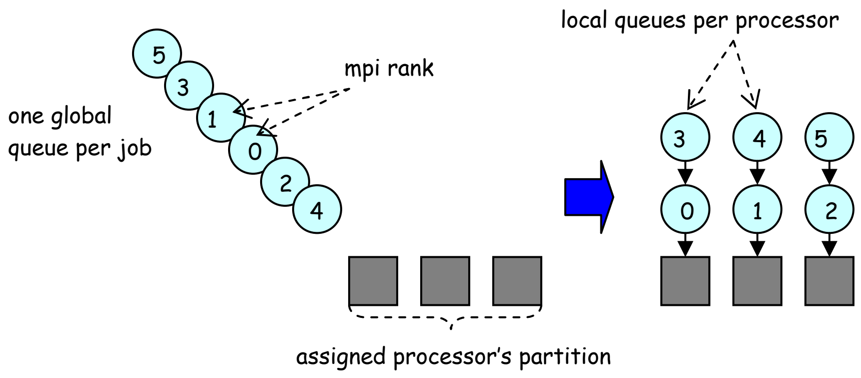


Figure 4.14 Process reorganization from a global queue to several local queues

4.4.5 Classification of applications applying LBD

This section is dedicated to present the results of applying the *load balancing detector* (*LBD*) mechanism to each of the applications evaluated in isolation.

Table 4.3 shows the result of the calculation of the average *IP* as well as the execution times of the applications when executed applying local queues per processor, global queues per application or the *LBD* dynamic mechanism of selecting the queue.

In the last section the number ***10%*** as the *IP threshold* limit to decide if an application has a well-balanced or imablanced behaviour , was established empirically.

Table 4.3 Execution times and average IP

	Local	Global	LBD	Avg IP
bt.A.36	298	342	306	3%
cg.B.32	390	521	408	6%
sp.B.36	193	248	195	6%
lu.A.32	135	151	140	9%
mg.B.32	54	80	58	4%
ft.A.32	23	24	25	4%
ep.B.32	49	51	51	3%
send.50x2	290	236	236	29%
send.50x3	437	315	315	31%
send.50x6	825	538	538	41%
bt-mz.B.36	447	368	368	59%

When working with well-balanced applications, it can be seen that the *LBD* adds a little overhead because they were running for a while under using global queues at the beginning of the execution.

4.5 Evaluations

In this section the experiments made to configurate the *virtual malleability*, as well as the evaluations made to compare the proposal of this chapter with other alternatives of the bibliography are described.

In order to evaluate *virtual malleability,* it has been considered three levels of scheduling proposals, according to the discrimination made in section 4.4. In each one it has been selected from the bibliography the alternatives that have demonstrated to perform best. All of them were implemented and evaluated.

4.5.1 Coscheduling policies evaluated

In this section the main characteristics of the policies evaluated from the bibliography and the ones proposed in this chapter are described.

Table 4.4 Coscheduling policies evaluated

	How to share processor	How to wait for a message	When to free the processor	Heuristic to select the next to run	Queue type
periodic boost	time-sharing	spin blocking	time-slice	unconsumed messages	local
IRIX	time-sharing	spin blocking	time-slice	round-robin	global
coscheduling: SB+RR	time-sharing	spin blocking	event-guided	round-robin	local
coscheduling: SB+Msg	time-sharing	spin blocking	event-guided	unconsumed messages	local
coscheduling: BI+RR	time-sharing	blocking immediately	event-guided	round-robin / sender	local
coscheduling: BI+Msg	time-sharing	blocking immediately	event-guided	unconsumed messages	local
virtual malleability: SB+RR	space-sharing	spin blocking	event-guided	round-robin	local
virtual malleability: SB+Msg	space-sharing	spin blocking	event-guided	unconsumed messages	local
virtual malleability: BI+RR	space-sharing	blocking immediately	event-guided	round-robin / sender	local / global
virtual malleability: BI+Msg	space-sharing	blocking immediately	event-guided	unconsumed messages	local

Table 4.4 shows the evaluated policies and their configuration according to the scheduling levels defined in section 4.5.2, such as processor sharing and processes scheduling on each processor.

The *periodic boost* technique was selected for being the *time-sharing* policy that has demonstrated in [NBSD99] to work best of the existing policies of *coscheduling*. It has been selected the native scheduler of *IRIX*, for being the operating system of the machine [SIGR00] where this work has been developed. The rest of the policies that appear in the table correspond to traditional policies of coscheduling and our proposal, the *virtual malleability* mechanism.

4.6 Performance results

This section is dedicated to show the performance results of the experiments for the evaluation of the proposals of this chapter.

The evaluations were divided in two steps. Firstly the *self coscheduling,* one of the proposals of this chapter is analyzed. The objective was to compare it to other coscheduling techniques in order to demonstrate the benefits of competing for the use of resources with the application itself instead of with others. The *self coscheduling* was evaluated with several configurations varying the type of actions taken related with communications events. The aim at this point was to deduce the best configuration when working with local queues.

After defining the best configuration for the *self coscheduling,* the whole mechanism of *virtual malleability,* under a more realistic environment was evaluated. It was analyzed with different *MPLs* and with workloads having well-balanced and imbalanced jobs.

The metrics used to do the comparisons were the response times of the workloads. In this work the response time of a workload is the time elapsed from the beginning of the execution of the first job of the workload till the termination of the last job of the workload.

4.6.1 Evaluation of coscheduling techniques

This section shows the performance evaluations of the techniques described in section 4.5.1 under closed workloads. All the applications arrive at the same time and begin their execution as soon as there are available resources.

As the sychronization between the processes of a parallel job is the main element in the performance of the applications, the workloads were constructed combining jobs with different communication degrees. The composition of the workloads can be seen in Table 4.5. Thus the workload *w1* is the one that has applications with greater communication degree of whereas *w4* is composed by applications with medium and low communication degree. The percentage of time dedicated to the execution of each one of the applications within the workload is equally distributed. The total number of applications per workload was 14 in order to have 10 minutes of execution approximately per workload. Each application was run with 64 processes. The workloads were run on a pool of 64 processsors.

About the way the traces of the workloads were generated, as well as the classification of the applications is described in chapter 3.

Table 4.5 Workloads composition varying the communication degree

workloads	applications	communication degree
w1	lu, cg, mg	high, high, medium
w2	cg, mg, ft	high, medium, medium
w3	mg, ft	medium, medium
w4	mg, ft, ep	medium, medium, low

The maximum MPL applied for the experiments of this section was four. This was the same applied in [NBSD99],[ZSMF00].

Figure 4.15 shows graphically the response times for the workloads described before. On the x axis are represented the workloads, and on the y axis, are the performance results expressed in seconds.

Workloads *w1* y *w2* are characterized by being composed mostly by high communication degree, point-to-point applications. Workload *w3* have mostly medium communication degree applications that perform collective operations and finally *w4* is composed by medium and low communication degree applications.

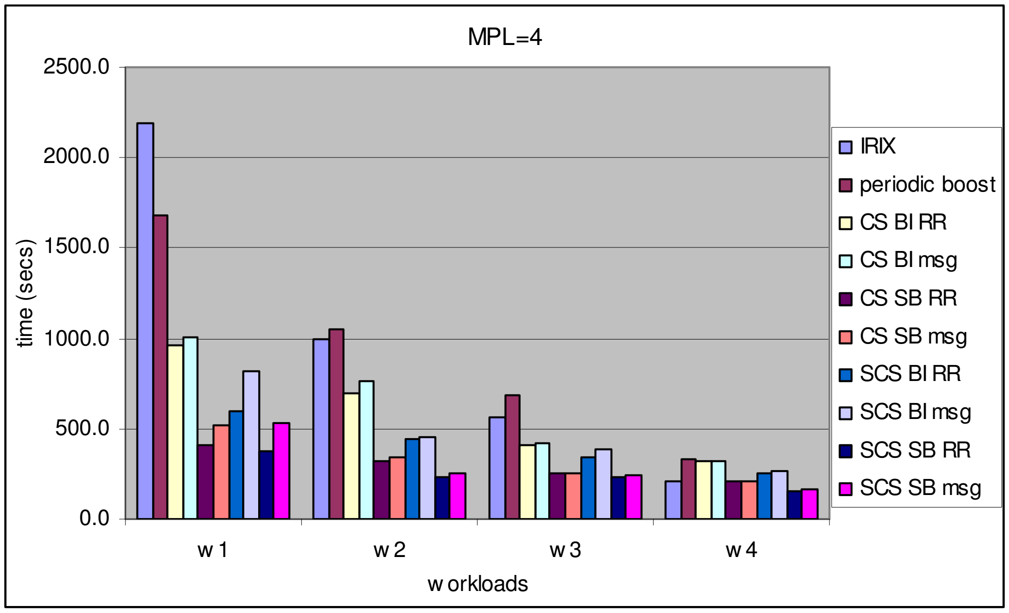


Figure 4.15 Response times for workloads with different communication degrees

As can be observed in Figure 4.15 the response times for the *periodic boost* and *IRIX* obtained the worst performance, especially in workloads with high communication degree.

The evaluations in [NBSD99] were done with 16 processors and 4 applications. Each application had 16 processes and the experiments were made on a workstations cluster. In preliminary evaluations of this work, it was reproduced the execution of those workloads obtaining similar results to them. So, it was deduced that when incrementing the number of processes and processors (64 in our case), the *periodic boost* is not flexible enough to keep synchronization with an acceptable performance. Whenever a message arrives it does not interrupt the process in execution, but its priority is increased. Each process frees the processor after finishing its time slice. So if a process is performing a blocking operation, it does busy waiting till the time quantum expires.

The *IRIX* scheduler does not take any special action when there are events related to messages. This scheduler performs local scheduling and hasn't got any mechanism of synchronization between the processes of the parallel applications. It does process migrations during the whole execution introducing overhead with the context switching and the loss of locality.

When waiting for a message, *blocking immediately* is the natural alternative for the platform that was used in this work. According to [DUCM98], the *spin time* must be calculated based on the latency of a roundtrip message across the network. They make their study on a network of workstations where the latency is not null. As the evaluations of this work were done on a shared-memory machine, this time tends to zero. Despite that, in this work there are experiments with spins times greater than zero obtaining the results shown above (SB).

On the selection of the next process to execute, it can be seen in the evaluations that *round robin* was the one that obtained better performance. It is a simple technique and it does not require any extra calculation like the number of unconsumed messages.

An interesting observation is that when applying *self coscheduling* (*SCS*), the applications execute always under the same conditions, no matter which are the rest of the applications of the workload. The application has the assigned partition for its exclusive use, and it has to compete for the use of resources with processes of the same application not with others.

On the other hand when pure *coscheduling (CS)* is applied, applications are more sensible with respect to the environment. This impact can be observed in the standard deviation of the execution times between different executions of the same applications within a workload. In Table 4.6 are shown the numbers that demonstrate that impact. For example the ***mg*** under *SCS* has coefficients of variation of the standard deviation between 5.5 and 7.8, while under the *CS* it has coefficients greater than 26.9. This means that *SCS* demonstrates to have more stability than *CS*. This is important in the sense that under *SCS* any application can have a predictable behaviour.

Table 4.6 Coefficient of the standard deviation (standard deviation / average execution time) under SCS and CS

	mg.B.64	ft.A.64	cg.A.64
CS SB RR	26.9	33.0	2.6
CS BI RR	38.2	17.7	2.9
SCS SB RR	5.5	4.0	2.7
SCS BI RR	7.8	12.2	1.7

The unpredictability of *CS* can be clearly seen when analyzing the execution of the *w4* workload. This workload is composed by high and low communication degree applications.

Table 4.7 shows the normalized response times for all the applications in workload *w4*. These were calculated by dividing the average response time of each application under *SCS* and *CS*, into the average response time of each application under a *FIFO* policy.

The *mg*, which is a medium communication degree application, obtained better performance under *SCS* than under *CS*. While the *ep* application, which performs calculation most of the execution time, obtained better performance under *CS* than under *SCS*.

This is due to that *CS* policies are unfair with respect to high communication degree applications when they must time share with low communication degree applications. This happens because a context switch is done only when there is no useful work to do, so if an application that do mostly calculations like the *ep*, will almost never free the processor degrading the performance of the other applications that share with it the resource.

Table 4.7 Normalized response times for applications within the w4 under SCS and CS.

	mg.B.64	ft.A.64	ep.B.64
CS SB msg	*8,1*	*2,6*	1,8
CS BI msg	7,2	2,3	**1,6**
CS SB RR	6,9	2,1	2,0
CS BI RR	6,6	2,0	**1,5**
SCS SB msg	3,7	2,0	2,9
SCS BI msg	3,5	**1,8**	2,8
SCS SB RR	3,2	**1,9**	3,0
SCS BI RR	**3,1**	**1,9**	2,9

4.6.2 Evaluation of the virtual malleability mechanism: self coscheduling + LBD

This section presents the performance results of the evaluations of the complete mechanism of *virtual malleability* and other *coscheduling* techniques which obtained the best performance in the last section. The evaluations were made under a more "realistic" environment. It has been considered several *MPL*s, different arrival times for the jobs and different system utilizations.

The workloads used are described in Table 4.8 and in Table 4.9. The first ones are composed just with well-balanced applications. The second ones have a mixture of well-balanced and imbalanced applications.

The arrival times of the jobs in each workload were generated following a Poisson distribution in such a way to reproduce machine utilizations of 60% and 20%. It has been considered workloads with mostly high communication degree applications and with mostly low communication degree applications. For a detailed description about how were generated the traces and classification of applications refer to chapter 3. The applicatios were run on a pool of 60 processors, leaving 4 for the execution of the CPUM, the launcher and performance analyzing tools [PARA01]. The number of processes assigned to each application was the maximum closest to 60: sweep3D was 60, the NAS except for the bt was 32, the bt was 36.

Table 4.8 Workloads composition varying the communication degree with well-balanced applications

workload	**high**	**low**
applications	bt, cg, mg, sweep3D	sweep3D, ep
communication degree	high, high, high, high	high, low

Table 4.9 Workloads composition varying the communication degree and with diffent balance degree

workload	**high**	**low**
applications	bt-mz, cg, send.50x2, sweep3D	bt-mz, ep
communication degree	medium, high, high, high	medium, low

In Figure 4.16 and in Figure 4.17 can be seen the performance results of the evaluations for the workloads shown in Figure 4.9 and in Figure 4.10 respectively, for 20 and 60%

machine utilization and *MPL*s 2, 4 and 6. In [MCFF98] they demonstrate that for a *MPL* greater than 6 is like applying infinite *MPL*.

The results are normalized with the response times obtained with the *FCFS* used as a reference.

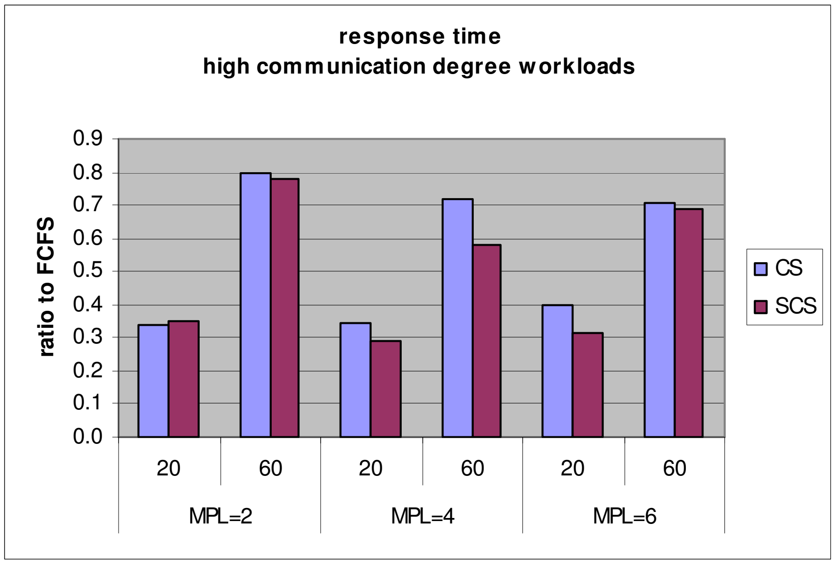


Figure 4.16 Response times for high communication degree workloads and well-balalanced jobs

SCS obtained better performance than *CS* especially on high communication degree workloads and with *MPL* equal to 4 in about 20%. When the *MPL* is equal to 2 and under low communication degree workloads, the SCS and CS policies obtained similar performance results.

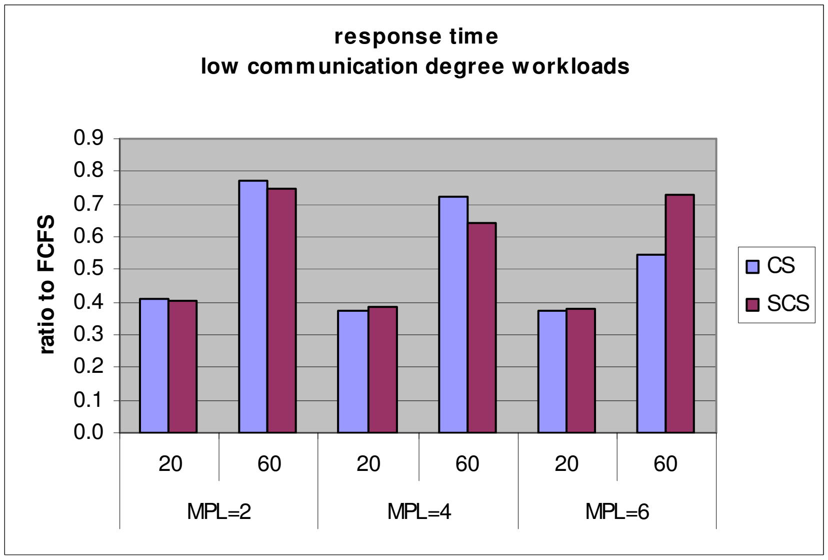
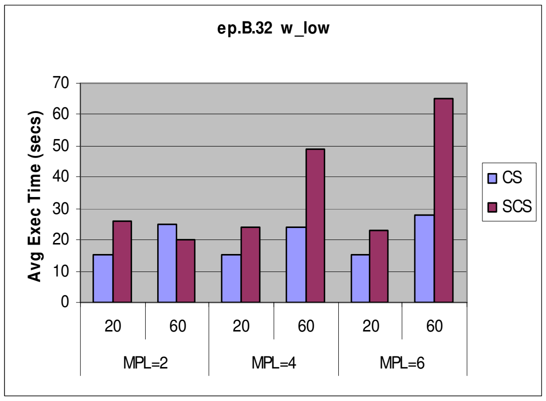


Figure 4.17 Response time for low communication degree workloads and well-balanced jobs

CS policies generate unfainess between the high and low communication degree applications when they share processor, favouring the second ones. This effect is shown when analyzing separately the individual performance of the applications that belong to the workloads.

In Figure 4.18 can be observed the average response times in seconds for the *ep* and the *sweep3D* applications, which belong to the low communication degree workload. The sweep3D obtained better performance under *SCS* than under *CS*. The *ep* releases the processor not very often, so it obtained better performance under *CS* than under *SCS*.



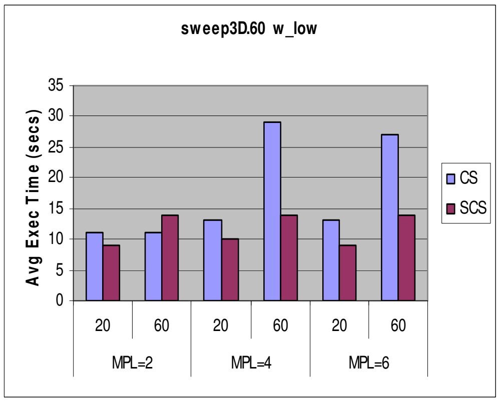


Figure 4.18 Average response time for ep (left) and sweep3D (right)

Next are shown the performance results for the workloads composed by well-balanced and imbalanced applications.

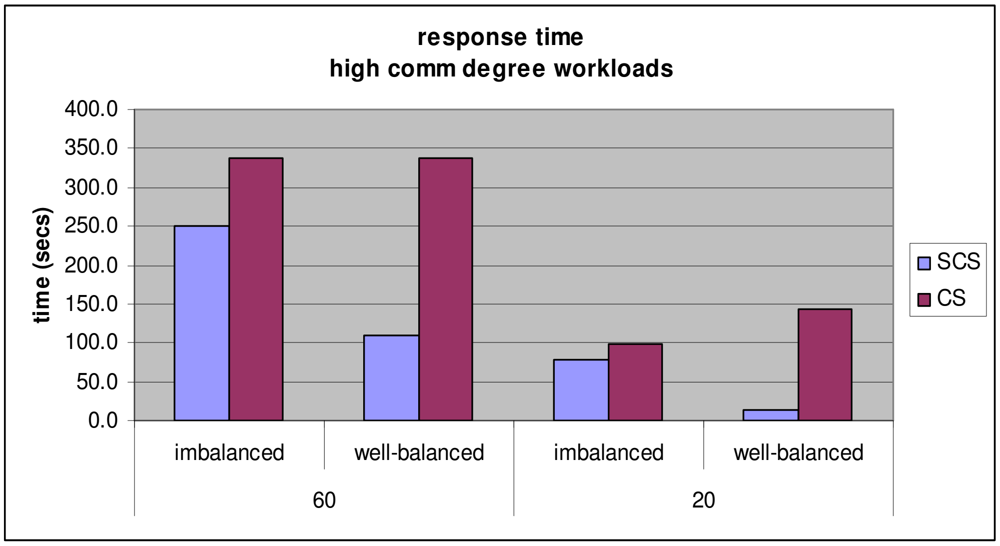


Figure 4.19 Response time for high communication degree workloads and different balance degree applications

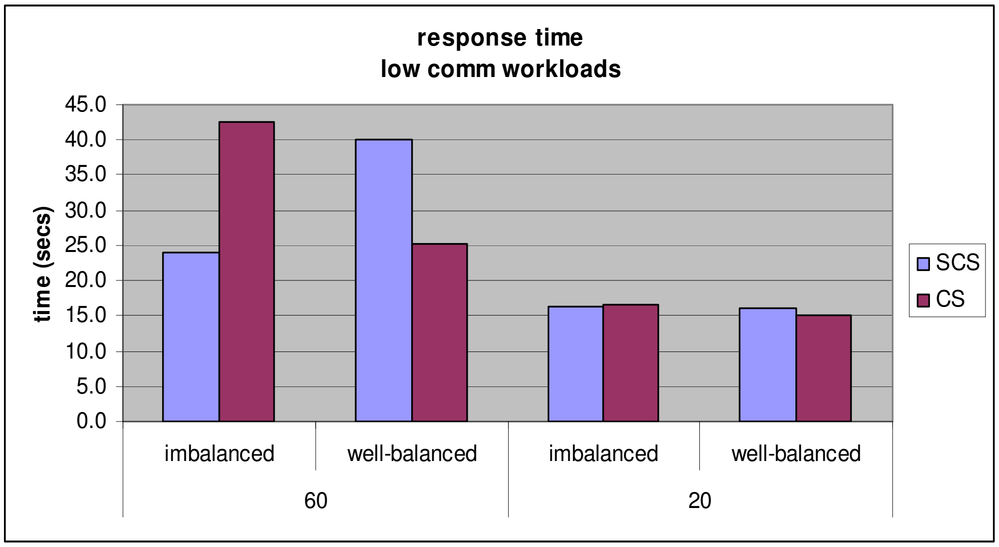


Figure 4.20 Response time for low communication degree workloads and different balance degree applications

Figure 4.19 shows the response times when the workloads are composed mostly by high communication degree applications with different load balance degree. It can be seen that when the application competes for the use of resources with itself (*SCS*) obtained better performance than competing with other applications (*CS*). This was true no matter the balance degree of the applications.

In addition, as the global queue approach was applied in the *SCS* case for imbalanced applications, the load balancing was done automatically. Under the *CS* techniques the processes were organized in local queues per processor, so the load balancing depended on the other applications with which shared the processor.

The response times when the workloads are composed mostly by low communication degree applications with different load balance degree are shown in Figure 4.20. It can be seen that for low machine utilization the performance was very similar for both scheduling techniques. However, as the machine utilization was incremented, the well-balanced applications obtained better performance under *CS* than under *SCS*. This is related with the effect commented before two applications with different communication degree share resources. The processor is most of the time being used by the low communication degree application, which rarely does context switching.

4.7 Summary

In this chapter the mechanism of *virtual malleability* was presented. This mechanism allows applications to adapt easily at runtime to the availability of the resources with the objective of improving performance and reducing fragmentation. It is composed by two techniques also described in this chapter: *self coscheduling* and *load balancing detector* (*LBD*).

Self coscheduling consists of assigning a processor partition to every application for its exclusive use. And as the number of processes could be greater than the number of processors, the technique forces the application to compete with itself for the use of resources. Processor sharing is done through coscheduling techniques.

About process organization, well-balanced applications showed better performance when using local queues per processor and imbalanced applications did it when using a global queue.

The coscheduling heuristics applied in the configuration of *self coscheduling* were selected after evaluating several alternatives related with how to wait for a message, what to do when the message arrives and, which process select next to execute.

When a process perform a receive message operation and the message is not available, the process is blocked immediately. In local queues the next process to be executed is selected in a round robin fashion, while in global queues is the sender process of the message that generated the blocking.

LBD classifies at runtime applications to as well-balanced or imbalanced, and after that applies the appropriate queue type to them.

Results showed that for high communication degree workloads and high machine utilization, *virtual malleability* demonstrated to have better performance than other coscheduling techniques from the bibliography. In addition, applications executed under this mechanism have a predictable behaviour, as they are executed in the same environment no matter how is composed the workload. This is because each application has its own processor partition, and has to compete with itself for the use of the resources. Under coscheduling techniques, applications have an unpredictable behaviour and are very sensitive with respect to the environment, as the processor sharing is done between processes from different applications.

Chapter 5
PROCESSOR ALLOCATION ALGORITHM: FOLDING BY JOBTYPE

Abstract

From the system's point of view, the virtual malleability facility becomes useful for jobs that are not able to modify their number of processes at runtime. In order to exploit this facility getting the maximum benefit, it is necessary to adjust some pertinent parameters at the beginning of the execution of each application. This chapter describes a processor allocation algorithm named Folding by Jobtype (FJT) which is in charge setting those parameters like number of processes and multiprogramming level, with the objective of maximizing system utilization and minimizing the response time of the applications in the system.

5.1 Introduction

Virtual malleability is a mechanism that allows a job to adapt to the current conditions in the system, incrementing the system utilization and minimizing the response time of the jobs in the system. In order to do that, the size of the processor partition of a job is modified at runtime.

Virtual malleability is achieved by modifying the multiprogramming level (MPL) of a job, so that it can be run in different partition sizes, with the same number of processes. Recall that MPL in this work refers to the number obtained when dividing the number of processes into the partition size.

From the system's point of view, this facility becomes useful for jobs that are not able to modify their parallelism dynamically at runtime, that is to say, they cannot modify their number of processes at runtime. This is the case for MPI jobs. So, in order to optimally use this facility and maximize its benefit, it would be interesting to adjust some pertinent parameters at the beginning of the execution of each application. Such parameters are the number of processes and its maximum MPL.

This chapter presents a processor allocation algorithm, *Folding by Job Type (FJT)* [UtCL1004], which forms part of the proposals of this dissertation. This algorithm is in charge of taking decisions related with the parameters before mentioned with the objective of maximizing the system utilization and minimizing the response time of the applications in the system.

Jobs are classified according to their sequential execution time. The type of jobs is provided by the users at submission time. The *FJT* algorithm takes into account the type and number of jobs that are in the wait queue and running, as well as their MPL. It decides for each selected job the number of processes, its maximum MPL and when to execute it.

The idea of the algorithm is to deduce from the available information, if more resources will become available. If so, an application with that is expected to be run for a long time, is applied virtual malleability. In this way it could start execution shrunk in a small partition, knowing that in the future it will be able to expand. On the other hand, applications that have short execution times start execution almost immediately by applying virtual malleability to longer execution time applications.

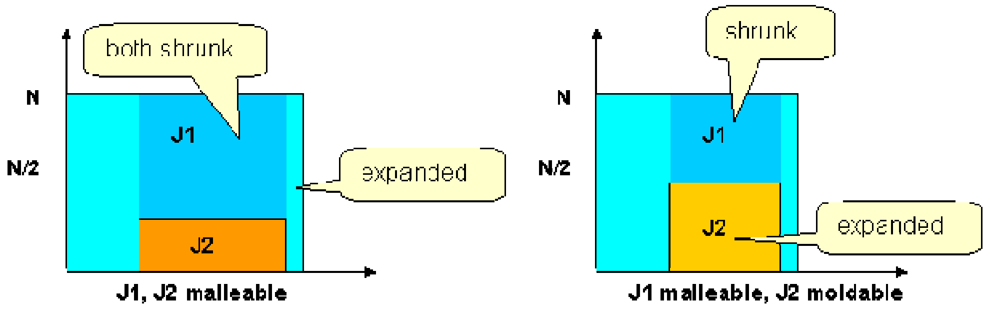


Figure 5.1 Virtual malleability applied to long and short jobs (left) and applied just to long jobs (right)

The *FJT* algorithm was implemented and compared to other processor allocation strategies from the bibliography. These are for example *folding* combined to moldability [PaDo96] or pure moldability [Cirn01],[RSSD99].

The results showed that *FJT* allows the jobs to adapt easily to the changes in the system load. The proposal obtained better benefits when the load of the system had dramatical changes, which was the case of burtly arrivals.

5.2 Related work

In [RSSD95] two moldability family techniques: *work-conservative* and *non-work-conservative* are described and evaluated. This classification is related with the decision about assigning all the available processors or explicitely keep some of them unassigned for future arrivals. The authors evaluate through simulations, ASP-MAX as a work-conservative strategy and PSA as a non-work-conservative strategy. In ASP-MAX the number of processes assigned to a job must be less or equal than the maximum parallelism the job accept or a constant MAX. In PSA, the number of processes of a job is calculated taking into account the number of jobs in the wait queue. If the number obtained is greater or equal to the number of the processors currently available, a non-work-conservative decision is made and the job doesn't start execution, leaving idle processors. This situation remains until new jobs arrive so the calculated partition will be smaller than before or a job finishes execution so the calculated partition will grow up. For the evaluation they use workloads with exponential arrival distribution time. They compare also different speedups. Finally they conclude that non-work-conservative policies obtain good performance when workloads don't scale well, have a high coefficient variation, there is a great variability between arrivals and are by bursts.

In [PaDo96], the authors apply Moldability to a technique proposed in [CaZa94] named Folding. Their objective is to reduce the queuing time. They create as many threads as processors there are in the system, then do multiplexing in order to execute the job in the reduced partition. Processors are shared by applying pure Time-Sharing. As the load increments, the latest arrived job is chosen and its partition is reduced to a half, freeing processors. For the evaluation they use synthetic applications with explicit synchronizing points in their code. The Folding must be done at these points, to explicitly do the process and data migrations. This generates additional wait time and requires extra effort from programmers and system support. The comparisons are against space-sharing with equipartition. The policy demonstrates to have an acceptable performance when load goes up. However, when load goes down, it is unable to get benefit from the new available processors.

Pure Backfilling [Lifk94] may seem a natural option. However as the jobs used in this work are assumed to be moldable and the workloads applied in this chapter have rather low machine utilizarion, there is almost no fragmentation so this technique has very little effect.

The objective of this work, apart from reducing queuing time, is minimizing execution time by taking advantage of the available resources generated when load varies, augmenting the overall performance.

5.3 Description of the Folding by Job Type (FJT) algorithm

In this section the *FJT* algorithm is presented. This algorithm is in charge of taking decisions that affect the whole execution of a job, as well as the general performance of the system. These decisions are concerned with the fact of applying *virtual malleability* and the parameters that are necessary to set at the beginning of the execution of each application.

In general, an irregular machine load will have execution peaks. If a job arrives during a high peak load, there would be none or few available processors. As recommended in the bibliography, the scheduler can just delay the execution until there would be enough free processors for it or start execution with just the few available processors. In the first case, wait time would become unacceptable and in the second case, it could happen that resources would become available later but the job will not be able to take advantage of them because MPI jobs are not malleable. Moreover if the job had a high execution time, it could increment considerably depending on the number of processes assigned.

The jobs are classified according to their sequential execution time as belonging to two classes: *long* and *short* which is shown in section 3. This information is provided by the user who submits the job. However, it doesn’t mean that the user must know its exact execution time.

This kind of classifications is commonly used in production systems, where a job is submitted to a different user queue depending on parameters like the estimated execution time and the number of required processors. The classification of jobs according to their sequential execution time is very simple, there are just two categories. However, this was considered enough to measure the impact when arriving jobs with different execution times and partition size requirements. In addition, the experiments done in this work were real executions, with exclusive use of the processors, thus introducing practical limitations to the duration of the experiments. On the other hand, it is common to find in the literature classifications of jobs with a number of categories between 2 and 4.

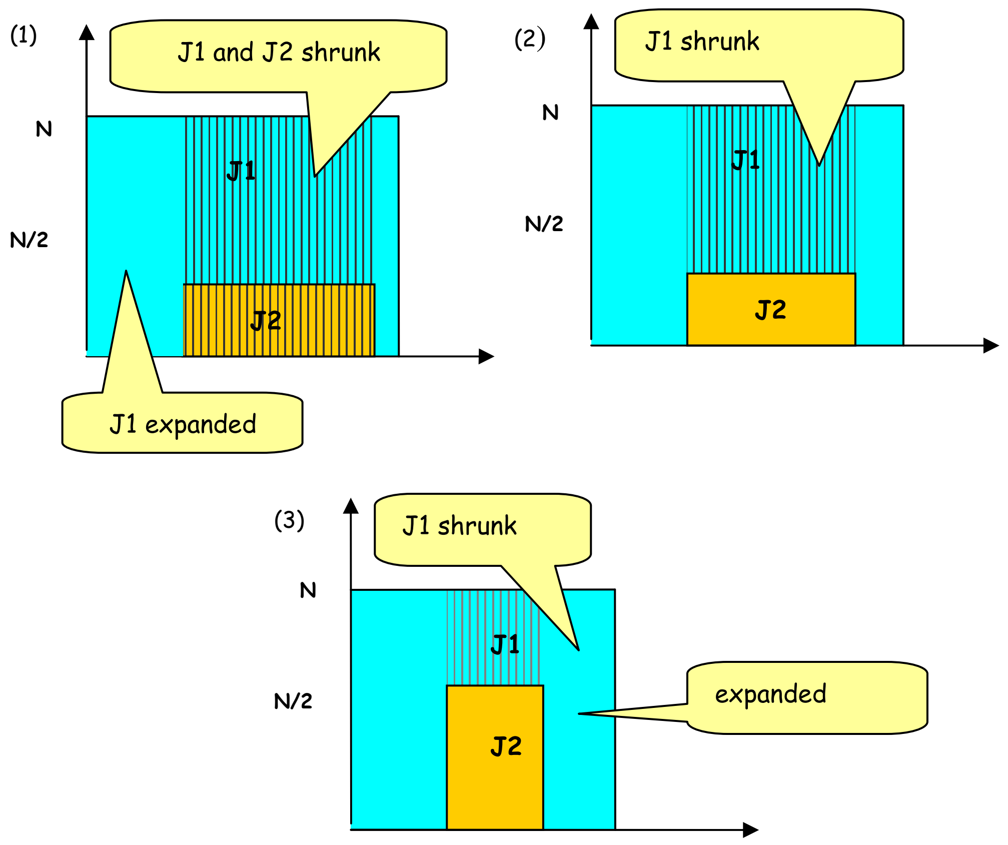


Figure 5.2 Scheduling alternatives when applying virtual malleability depending on the flexibility of the jobs

Examples of the execution of two jobs, one short (J2) and one long (J1) are presented in Figure 5.2. They are executed under different scheduling alternatives when applying virtual malleability. In (1), virtual malleability is applied to both jobs. After starting J1 its execution, J2 arrives and both are executed in a partition with size smaller than their number of processes. As J2 is a short job, it finished execution before J1 did. J2 job was all the time executed shrunk, so it never had the opportunity to expand. In (2) virtual malleability is applied just to J1. When J2 arrives it starts execution with a number of processes equal to its partition size. In this scheme J2 obtained a better performance than in (1) due to the fact that it was all the time executed expanded, thus eliminanting the overhead of being shrunk. In (3) virtual malleability is applied just to J1 as in (2). But in this case the size of the partition assigned to J2 is greater than in (2) so it obtained a better response time than in (2). In addition J1 was executed shrunk a shorter time than in (2) so it obtained a better performance as well. Alternative (3) corresponds to *FJT* algorithm.

In conclusion it is necessary to know in advance the type of jobs before making any decision in order to "predict" the near future.

In Figure 5.3 the mechanism that implements the *FJT* algorithm is presented graphically. Whenever there is a change in the availability of the resources, as a result of a job termination, or the wait queue goes from empty to not empty, an event is triggered and *FJT* is applied. The available information to the mechanism is the number and type (*long* or *short*) of the jobs in the wait queue and currently running, the current system state and the maximum MPL allowed for each running job.

The algorithm needs extra information about the jobs, that is to say, if it is *long* or *short* according to the classification made in section 3.1.7. This information is provided by the user when it submits the job for execution. This kind of classifications is commonly used in production systems, where a job is submitted to a different user queue depending on parameters like the estimated execution time and the number of required processors.

The classification of jobs according to their sequential execution time is very simple, there are just two categories. However, this was considered enough to measure the impact when arriving jobs with different execution times and partition size requirements. In addition, the experiments done in this work were real executions, with exclusive use of the processors, thus introducing practical limitations to the duration of the experiments. On the other hand, it is common to find in the literature classifications of jobs with a number of categories between 2 and 4.

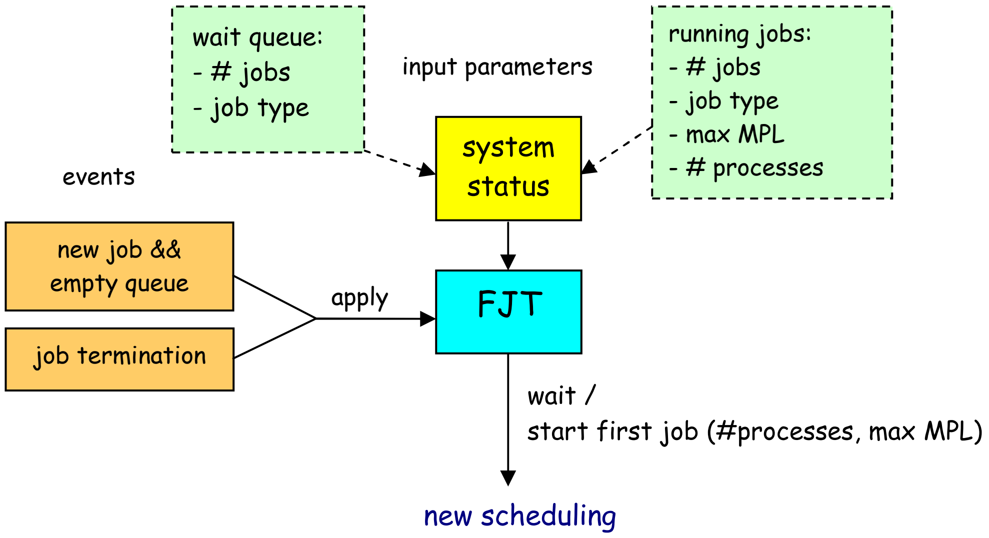


Figure 5.3 Mechanism that implements the algorithm of FJT

Whenever possible the launcher tries to dispatch the job at the head of the queue applying *FJT*. This algorithm has to decide: 1) execute now or later; 2) maximum MPL; 3) number of processes. In order to take such decisions, the algorithm takes into account all the available information from the system, including the wait queue. So if the minimum requirements are satisfied, the job is launched for execution; otherwise the job is kept in the wait queue.

If the first job from the wait queue is *long*, it could be scheduled if one of the conditions applies: 1) there are short jobs in execution and their processors are not expected to be assigned to currently running *long* jobs; 2) the wait queue is empty. Then the job is assigned a number of processes bigger than the size of the available partition. This *long* job will be executed shrunk till a short job finishes execution. After that the *long* job is able to expand to the newly available resources. It is important to notice that the waiting time of the job was reduced even the job had to be executed shrunk for a while. On the contrary, if the first job from the wait queue is *short* and there are no free processors, the *long* jobs are shrunk temporally freeing resources for the short one.

So taking all this into account and applying virtual maleability only to long jobs, there are a lot of possible combinations, which were treated separately in order to design the algorithm. Figure 5.5 shows a pseudocode for the *FJT* algorithm.

About the size of the partition to assign initially to a job is done in the following way: if there are jobs in the wait queue, an equipartition between all of them is done like in PSA [RSSD99]. If there were long jobs in the wait queue, the equipartition is done just between them. This is because this kind of jobs is able to shrink and expand each time a short job start and finish execution respectively.

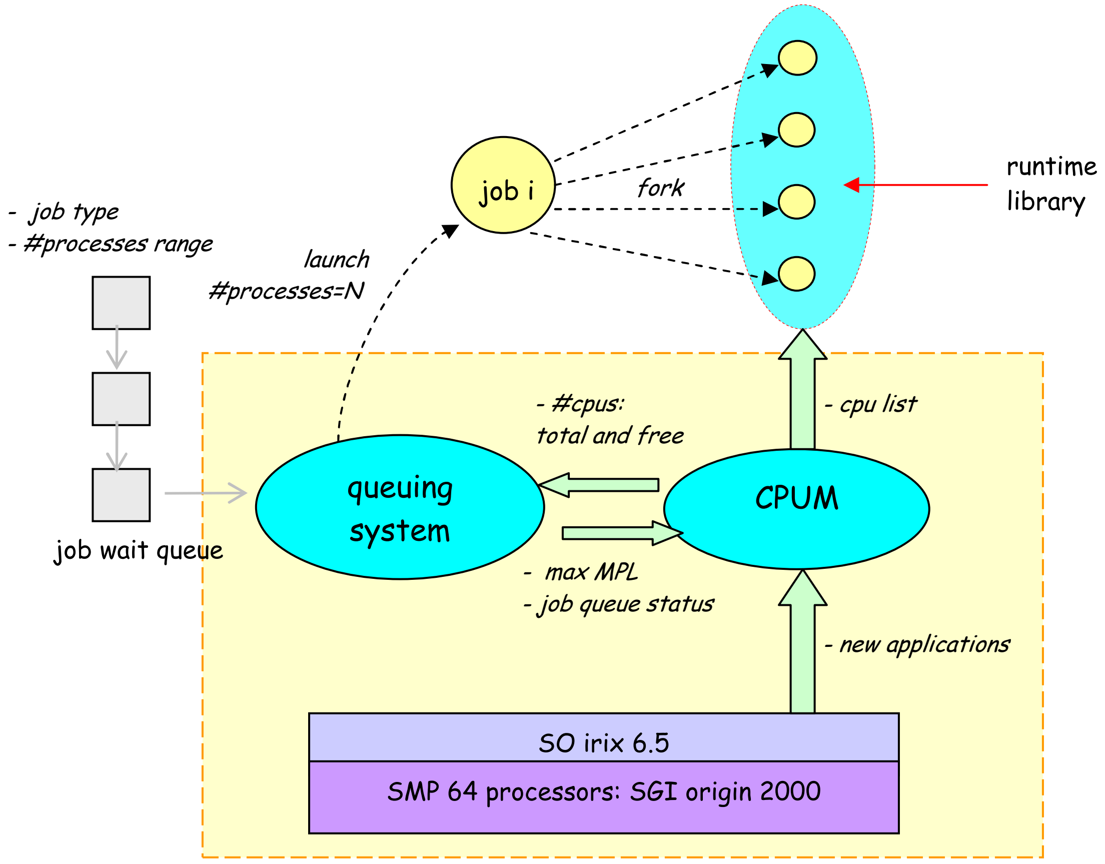


Figure 5.4 Execution environment related to the algorithm FJT

Figure 5.4 shows the system components that implement the *FJT* algorithm. This algorithm forms part of the queueing system, the *launcher*, which in coordination with the *CPUM* applies the job scheduling policies. The *CPUM* provides the launcher with information about the available resources.

```
If First Job = Short =>
              If there are idle processors or
            long jobs able to be shrunk Then Exec
        Else Wait
        End If
Else if First Job = Long =>
        If there aren't long queued jobs then
           If there are short jobs running Then Wait
              Else If there are idle processors Then Exec shrunk
              Else if long jobs running expanded Then Exec shrunk
              Else Wait
           End If
  Else /* there are long jobs queued */
              If there are idle processors or
               long jobs running expanded Then Exec expanded
          Else Wait
              End If
  End If
```

Figure 5.5 FJT algorithm

FJT sometimes takes work-conservative decisions and sometimes non-work-conservative ones. This means that sometimes all idle processors are assigned to the newly arrived jobs, and there are situations when some processors are kept idle even though the wait queue is not empty. For example if the first job is a long one

5.4 Evaluation

This section is dedicated to the description of the evaluated policies as well as the performance evaluation experiments.

5.4.1 Policies evaluated

These are the processor allocation policies selected for the evaluation of the proposal of this chapter:

a) *folding* with moldability: It reduces the partition to a half everytime there are not enough resources for the first job in the wait queue. It also applies moldability.

b) *PSA:* This policy is described in the related work section. It is a *non-work-conservative* moldability technique, which means that all idle processors are assigned.

c) *ASP-MAX:* This policy is described in the related work section. It is a *work-conservative* moldability technique, which means that just a certain percentage (*MAX*) of the idle processors are assigned. The *MAX* constant was fixed in 60% as it was the one that obtained better performance.

Table 5.1 shows the main characteristics of the evaluated policies, as well as a comparison from the point of view of processor allocation.

Table 5.1 Comparison of the main characteristics of the evaluated policies

	PSA	ASP-MAX	FOLDING	FJT
Limit # Processors	ncpus	MAX	ncpus	ncpus
Job classification	no	no	no	yes
Queued jobs	equi-partition	-	-	equi-partition
Work-conservative decisions	no	yes	no	yes
Non-work-conservative decisions	yes	no	yes	yes
Folding	no	no	yes	yes
Initial Folded times	-	-	1	4 (*long* jobs)
Maximun MPL	1	1	N	4 (*long* jobs)
Space-sharing	yes	yes	yes	yes
Processor Scheduling	-	-	pure time-Sharing	virtual malleability

The *ASP-MAX* policy states a maximum number of processors to allocate while in the rest of evaluated policies this number is just limited by the total number of processors of the system (ncpus).

About queued jobs *ASP-MAX* and *FOLDING* do nothing. *PSA* does an equipartition between all the queued jobs. And *FJT* does an equipartition between long lobs in case there are any; otherwise the equipartition is between all the queued jobs. *PSA* and *ASP-MAX* have always their maximum MPL set to 1, while *FOLDING* doesn't have a limit and *FJT* is set to 4. Notice that while *FOLDING* initially start their jobs with MPL set to 1, the *FJT* may start long lobs with MPL set from 1 to 4.

5.4.2 Performance results

In this section are presented the performance results obtained from the evaluation of the policies before mentioned.

The arrival times for the jobs in the workloads used for the evaluations were generated by applying the equation in chapter 3. The composition of the workloads was determined by combining applications with different sequential execution times, that is to say, using the classification of long and short jobs. This is because in this chapter the main objective is to analyze the impact on the performance of the algorithm proposed when jobs with different execution times arrive.

The applications that take part on the workloads are shown in Table 5.2. The workloads were built by varying the proportion of *long* and *short* jobs. The workloads were adjusted to last between 600 and 900 seconds. Each one is composed by 50 jobs approximately. The number of processes chosen by each application can be different in each experiment, because of the FJT algorithm and the environment context at the moment the application is launched.

Table 5.2 Applications used in the workloads

workload	**long**	**short**
applications	bt.A, cg.B	bt.W, sweep3D
communication degree	high, high	high, high

The evaluations were made for average machine utilizations of 50%, 60% and 70%. It was considered these medium machine loads because the *virtual malleability* mechanism has sense only when the load varies and is not saturated all the time.

Table 5.3 presents the composition of each workload in terms of machine utilization of *long* and *short* jobs.

Table 5.3 Long and short jobs relation within a workload.

	% machine utilization	**% utilization long Jobs**	**% utilization short Jobs**
w1	50	40	10
w2	60	40	20
w3	70	40	30
w4	50	10	40
w5	60	20	40
w6	70	30	40

The performance results for the policies described in section 5.5.1 under the workloads of Table 5.3 are summarized in Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9. First are shown the results for the workloads *w1*, *w2* and *w3*, where the workloads are composed mostly by *long* jobs. *Short* jobs have machine utilization between 10% and 30%. Then are presented the results for the workloads *w4*, *w5* and *w6* where most of the machine utilization is done by *short* jobs. *Long* jobs have machine utilization between 10% and 30%.

Figure 5.6 shows the average response time for *long* jobs in *w1*, *w2* and *w3* workloads, detached in average waiting time and average execution time. As it can be observed *FJT* obtained the best response time. Another interesting detail is that its performance keeps constant even though the machine utilization changes. The execution time of the jobs is the component that has greatest impact on the performance of *long* applications.

On the other hand, for short jobs, the waiting time is the component that showed the greatest impact on the performance for all the policies evaluated except for *FJT*, as can be observed in Figure 5.7. As the load increases, *FJT* degrades the performance of *short* jobs favouring the *long* jobs while *PSA* favours the performance of *short* jobs. This is because *PSA* distributes processors with an equipartition between all the jobs in the wait queue, no matter if they are are *long* or *short*.

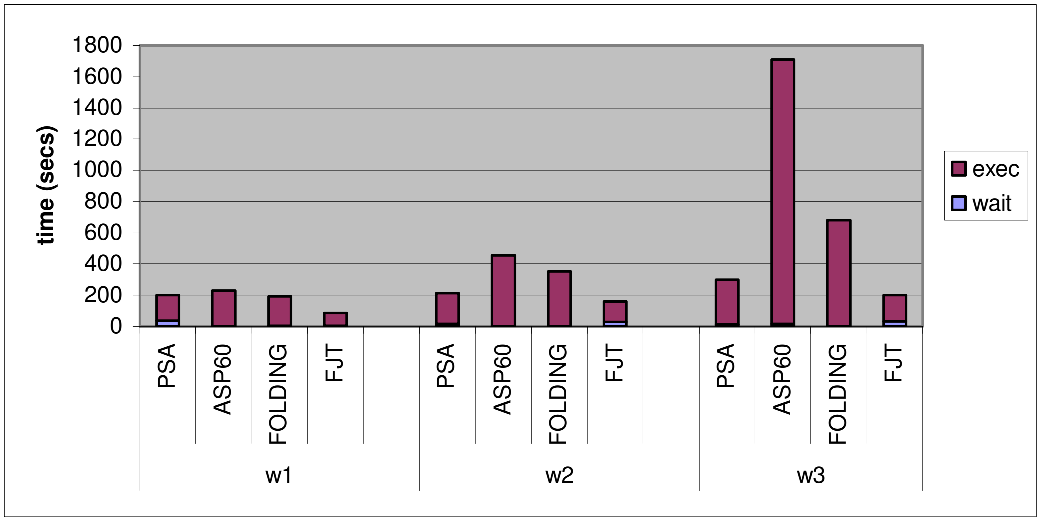


Figure 5.6 Average response time for long jobs (40% long jobs, 10-30% short jobs)

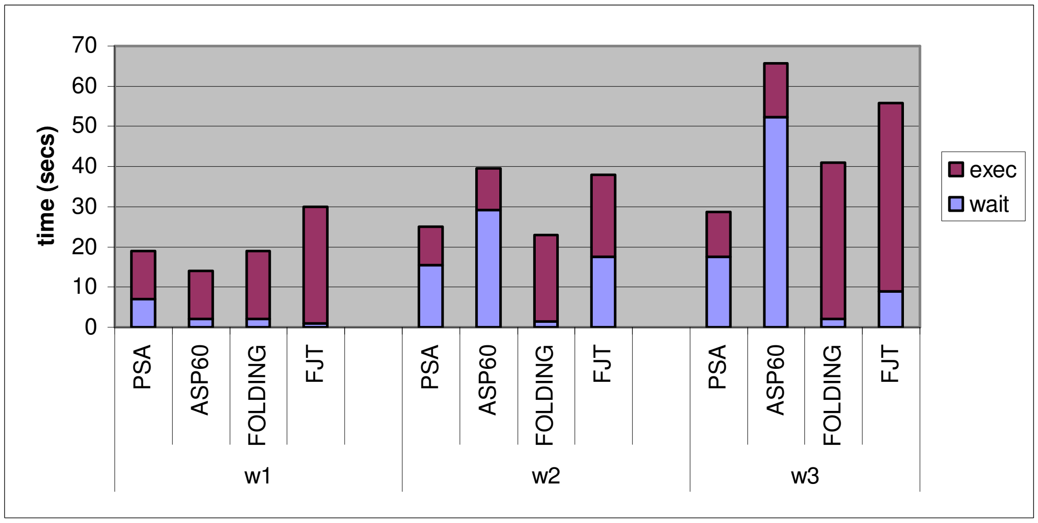


Figure 5.7 Average response time for short jobs (40% long jobs, 10-30% short jobs)

Next are the performance results for the workloads *w4, w5* and *w6*, where 40% of the machine utilization is dedicated to *short* jobs and from 10 to 30% is dedicated to *long* jobs.

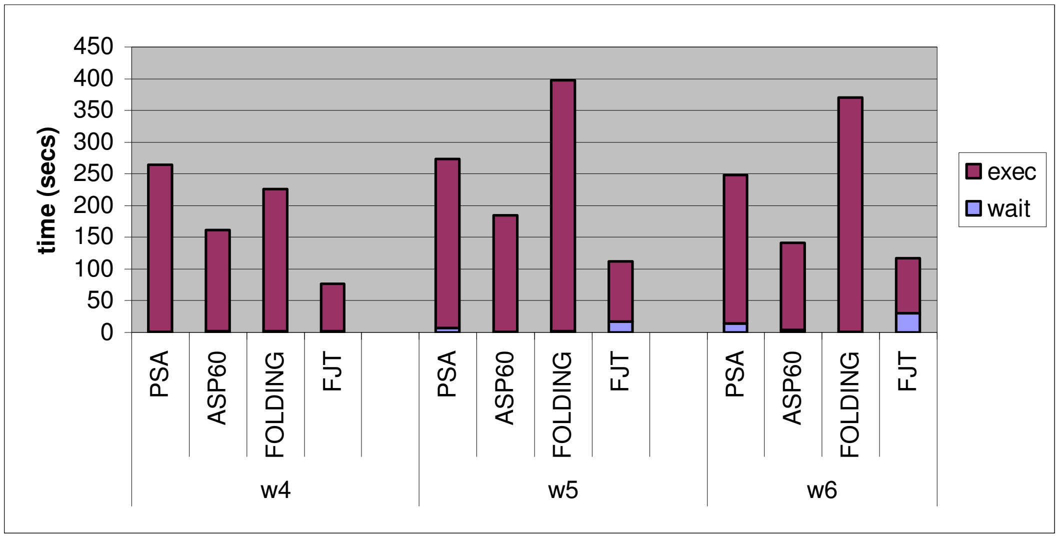


Figure 5.8 Average response time for long jobs (40% short jobs, 10-30% long jobs)

Figure 5.8 shows the average response time for *long* jobs detached in average waiting time and average execution time. It is possible to observe that *FJT* obtained the best performance and more notoriously than in the previous evaluation. As a matter of fact, the relation between the policies is different from the previous evaluation. Folding and *PSA* obtained the worst performance. Let's analyze what happened. In these workloads the total number of jobs is greater than the other ones. This is because, even the total machine utilization is the same, the *short* jobs have greater proportion, and as they are short, to increment their machine utilization, it was necessary to increment their number. In this context when *PSA* does the equipartition, the result is a smaller partition size degreading the performance of *long* jobs dramatically.

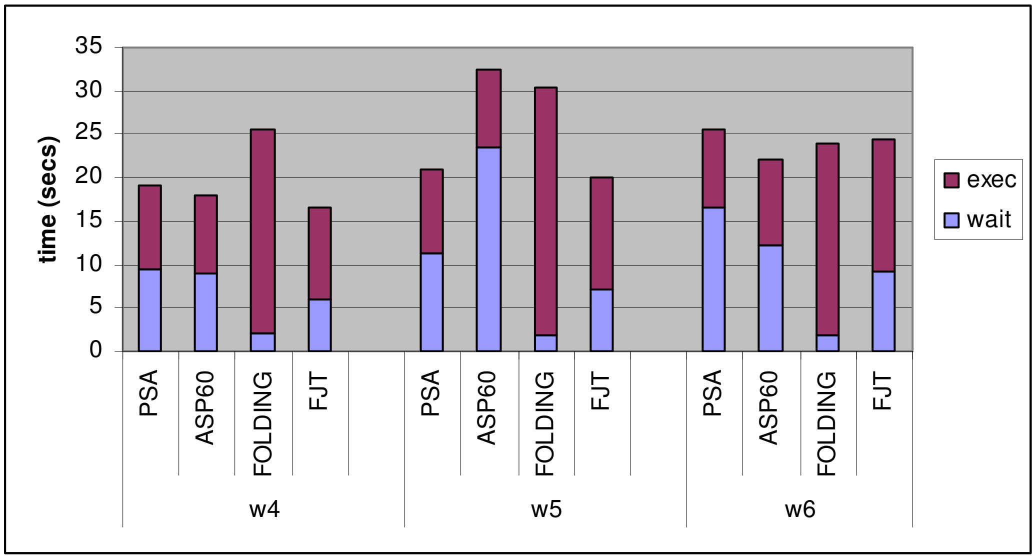


Figure 5.9 Average response time for short jobs (40% short jobs, 10-30% long jobs)

The folding policy doesn't differentiate between *long* and *short* jobs. As jobs are folded, their performance is degraded especially *long* ones. Despite that, it achieves its objective of reducing the waiting time.

5.5 Summary

In this chapter a scheduling algorithm at queueing system level, named *Folding by Job Type (FJT)* was proposed. The algorithm takes some decisions concerned with the number of processes, the maximum multiprogramming level (*MPL*) and when to execute the jobs in the wait queue.

The jobs are classified according to their sequential execution time. The algorithm is based on applying virtual malleability to jobs that have high execution time and moldability to jobs that have short execution time. *Long* jobs can be run shrunk temporally to reduce wait time or to allow the execution of *short* jobs.

FJT differs from other previous processor allocation strategies in its aggressiveness which allows the jobs to adapt easily to the changes in the load, taking advantage of the temporal available resources.

The proposed algorithm was implemented and compared with others processor allocation policies like folding [CaZa94] with moldability [PaDo96] and two moldability family techniques *ASP* and *PSA* [RSSD99]. They were evaluated under workloads with different machine utilization from low to medium, and different proportion of *long* and *short* jobs. The workloads used for the evaluations had low to medium machine utilization in average. This decision was made because, *FJT* consist on taking advantage of temporal freed resources which cannot occur in a machine with high machine utilization all the time.

Performance results showed that *FJT* can adapt easily to the changes in the load. It obtained the best performance especially for *long* jobs in about 30%. For *short* jobs the performance was similar and in some cases worse than the rest of the evaluated policies. The objective of the *FJT* is to minimize the response time, not only the waiting time is it is the case for the folding policy.

Chapter 6
FOLDING BY JOBTYPE with BACKFILLING

Abstract
This chapter presents the algorithm FJT combined with the backfilling techniques in order to alleviate fragmentation generated when working with heavy loaded machines.
The effectiveness of the backfilling techniques relies on user time estimations. This chapter proposes an alternative when the backfilled jobs expire their execution window time: instead of aborting or suspending them, virtual malleability is applied, thus freeing resources.

6.1 Introduction

In the previous chapter it was presented an algorithm that defines a processor allocation and job scheduling policy, *Folding by Job Type* (*FJT*), which applies the idea of *virtual malleability* to long jobs and moldability to short ones. A job is selected from the the wait queue and taking into consideration the general state of the system, the algorithm takes decisions concerned with the number of processes, the maximum *MPL* and when to execute the job, with the objective of minimizing response times and incrementing machine utilization.

What is proposed in this chapter is [UtCL0605]:

1. Add the backfilling techniques to the *FJT* algorithm.
2. Add the virtual malleability mechanism as a strategy for the problem of expired windows of backfilled jobs.

This means that the backfilling techniques are added to the *FJT* algorithm. And whenever a backfilled job has its execution window expired, virtual malleability is applied to it. In this way, the backfilled jobs are neither aborted nor suspended and the first job in the queue is not delayed either. The partition size of the backfilled job is reduced freeing resources and the first job in the queue can start execution. If the backfilled job were aborted it would had freed more resources, but as the jobs are supposed to be moldable, they could adapt easily.

It could be posible that even applying virtual malleability to backfilled jobs and thus reducing their processor partition size, there were not enough processors left for the job at the head of the queue. However, this case would not very common as the jobs are moldable and could adapt to the available resources.

As a result, the backfilled jobs that have their execution window expired are able to continue execution minimizing their wait queue time even when the user time estimation was incorrect.

Figure 6.1 shows the difference between traditional backfilling and the proposal of this chapter. It can be observed two executions with moldable jobs and applying the backfilling techniques. The sequence of execution is the following: *job 1* started to execute, then arrived *job 2* but the available partition was not enough for it so it had to wait till there were more available resources. *Job 3* is backfilled to fill the gap generated. The window time of *job 3* is limited to the execution time of *job 1*. When *job 1* finished its execution, the window time of *job 3* expired. In the execution of the left (a), it is applied aggressive backfilling, where backfilled the job is aborted and reinserted in the wait queue. In the execution on the right (b), it is applied virtual malleability to *job 3* thus reducing its partition size.

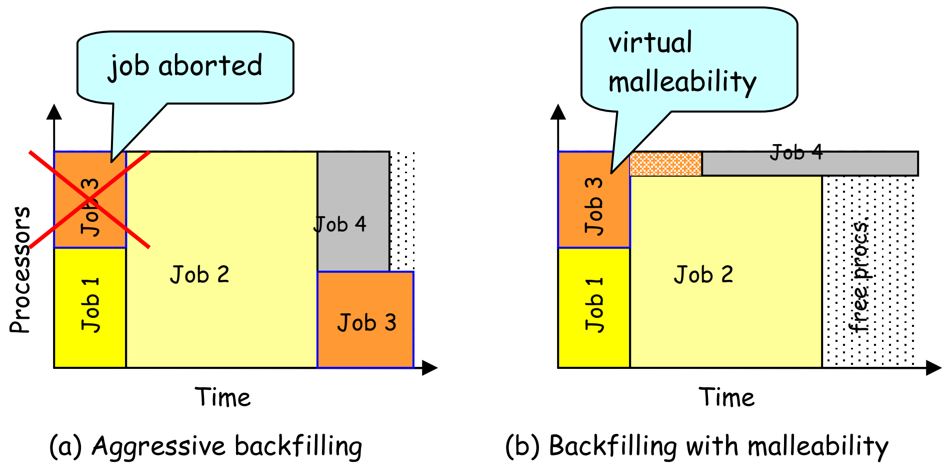


Figure 6.1 Traditional backfilling (left) and backfilling with malleability (right)

The proposal was implemented and compared to *FJT* with different MPLs and several backfilling alternatives. The results showed a performance improvement up to 25% over traditional backfilling with high machine utilization.

The addition of virtual malleability to the backfilling techniques reduces the overhead generated by aborts or suspensions, as well as it prevents from reinserting the backfilled jobs in the wait queue. These jobs would become eligible to be backfilled again thus wasting resources.

6.2 Motivation

Virtual malleability reduces processor fragmentation by adapting the size of the processor partition assigned to jobs to the available resources at runtime. However, as the machine load increments the performance of the system degrades. This is due to several reasons: 1) Errors in the prediction on the load of the system; this leads an application to be executed totally or parcially shrunk all the time. 2) The number of times an application is shrunk and expanded; as this number increments it generates certain overhead, eliminating the gain when it was executed expanded. For the moment this number is not taken into consideration in the algorithm of *FJT*. It depends strongly on the application being scheduled, the impact of loosing memory locality. In order to improve the prediction it should be necessary a more complex mechanism that could learn from the application at runtime such characteristics. And this is out of the scope of this work.

Under high machine utilization workloads, applications have MPL set to 1, which corresponds to the moldability concept. However, not all the applications accept any number of processes. As an example *bt* and *cg* from the NAS benchmarks just accept perfect squares and power of 2 sizes respectively.

In this context processor fragmentation is generated when applying moldability as the unique way to adapt to the available resources and because of the errors of the prediction. To alleviate the fragmentation, usually systems apply the backfilling techniques [Lifk94].

These techniques consist on moving jobs ahead in the queue, provided that they do not delay any previous job in the queue. Backfilling techniques have demonstrated to improve system utilization and to reduce job wait times versus the same policy without backfill [WeFe01].

This approach depends on user time estimate for its effectiveness. In [WeFe01],[ZFMS00] they state that overestimation has little impact on the performance of such policy. However if the execution time of a backfilled job is underestimated, some action has to be taken with the backfilled job: 1) abort [SnCJ02]; 2) suspend/resume; 3) checkpoint/restart; 4) remain executing delaying the rest of jobs in the queue [TaFe99][WaMa02].

Except for option 4), the scheduler has to reinsert the backfilled job in the wait queue. In option 2) the job must be resumed in the same processor partition, unless it is running on a shared-memory multiprocessor, in which case it would be also advisable to minimize the memory impact. This may add a considerable delay for resuming the job. In addition not all the operating systems have support for option 3).

6.3 Related work

In order to avoid starvation of jobs, conservative backfilling requires that the execution of the backfilled job does not delay any job arrived earlier [WeFe01]. Aggressive (EASY) backfilling relaxes the condition, allowing backfilling jobs whenever they do not delay just the job at the head of the wait queue [Lifk94]. In this strategy they will be allowed to backfill more jobs than in the previous one, since it is simpler to avoid the delaying a work that all in the queue. There isn't any consensus about which of the proposals is better. For jobs that request many resources are better conservative backfilling, whereas for jobs that are short it is better aggresive backfilling.

There are several variations to backfilling techniques proposed in the literature. In [TaFe99] Talby and Feitelson, proposed to incorporate a priority system to eliminate the possible starvation of jobs introduced by aggresive backfilling and to remove the rigidity in moving jobs ahead given by conservative backfilling. The priority of each work is provided by the user, the policy and the scheduler. This one can be modified depending on the elapsed time in the wait queue. In their simulations based on a year trace from a production system, they obtained a benefit of 15% over conservative backfilling. However, this gain is strongly depended with their calculation of priorities.

Srinivasan et al. present in [SKSS02] a selective backfilling wherein jobs do not get a reservation until their expected slowdown exceeds some threshold. It pretended to be an intermediate approximation between conservative and aggressive backfilling. In their simulations this strategy obtained an acceptable performance when the user time estimations are perfect. They used a maximum MPL equal to 2. This strategy is a variation of [TaFe99], with a simpler calculation of the threshold. The results are strongly depended to this number.

There is an interesting work presented in [ShFe03] about which job choose from the wait queue. Shmueli and Feitelson proposed the use of dynamic programming to look deeper into the queue and select a set of jobs, which together would maximize the machine utilization.

In [FFFP03] Frachtenberg et al evaluate the impact of adding backfilling techniques to gang scheduling [Feit97] and to flexible coscheduling [FFPF03].

Lawson and Smirni presented a multiple-queue backfilling approach, where each job is assigned a queue and a partition depending on its estimated execution time [LaSm02]. In their simulation results they obtain a gain in performance with respect to a single queue-backfilling.

About integrating moldability and backfilling techniques in [SSKH02] Srinivasan et al. propose a technique that selects the partition size for a job based on its scalability and turnaround time by applying the Downey model [Down97]. They add aggressive backfilling and demonstrate a gain in performance over pure backfilling and pure moldability [Cirn01].

6.4 FJT with Backfilling

The proposal of this chapter was constructed from the job scheduling algorithm *FJT* presented in the last chapter. Backfilling techniques are added to the mechanism, as well as a new way of treating the backfilled jobs which have their window time expired.

6.4.1 FJT in high loaded systems

FJT takes advantage of changes in the load, especially when it goes from high to low peaks. It showed an acceptable performance in workloads from low to medium average machine utilizations.

But as the load increments, the performance of this algorithm degrades because the applications have almost no opportunity to expand and take advantage of occasionally freed resources. This generates overhead as the applications executes shrunk in the smallest possible partition which is not compensated with the number of times they are able to expand and loosing locality. So in this context it is not worthy to assign MPLs greater than 1 to any job.

The proposal of this chapter is centered on high loaded machines, where there is usually a queue of jobs waiting to be executed. The maximum MPL assigned to any job is 1, which means that applications are treated at most as moldable.

6.4.2 Adding backfilling to FJT and virtual malleability to expired windows

Applications are not always completely moldable, it means, not all the applications accept any number of processes. In addition a *long* job should not be assigned a small number of processes, which could result in an unacceptable performance. So there are new sources of fragmentation. In order to alleviate the fragmentation, in this chapter is proposed to enhance *FJT* with *backfilling* techniques [Lifk94].

Backfilling is based on moving ahead jobs in the wait queue to fill gaps generated by the fragmentation. In order to backfill the jobs, it is necessary to have an estimation of its execution time. Those estimations are provided by the user, which could lead to overestimations and underestimations. The overestimations had little effect as demonstrated in [ZFMS00].

In summary, whenever the job at the head of the queue could not be executed because the available resources are not enough for it, the launcher tries to backfill a job from the wait queue. The jobs that are eligible to be backfilled are the *short* ones and that could be assigned a number of processes that fits in the available partition.

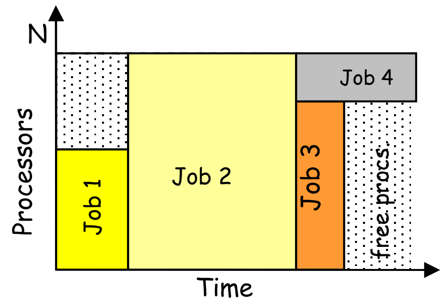
6.4.3 Virtual malleability applied to expired windows

About the underestimations, if the window time of a backfilled job expires and it didn't finish its execution, instead of for example aborting or suspending it [Lifk94][TaFe99][WaMa02], it is proposed to apply virtual malleability to it. This is the only case where virtual malleability is applied to jobs in this new proposal.

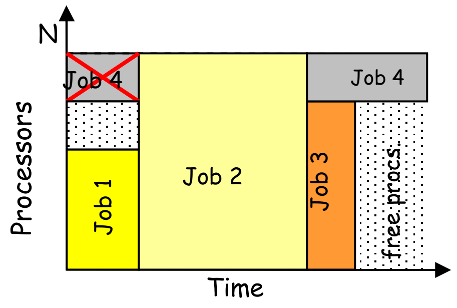
In this context an execution window is said to be expired when all the jobs that are currently in exection are out of order, this means, all of them have arrived after than the one at the head of the queue. This is a more relaxed condition, than the one in aggressive or conservative backfilling. In addition the user time estimations are just the class each job belongs to.

Every time an execution window expires and the backfilled jobs haven't finished execution, the launcher applies virtual malleability to them, reducing their partition to the minimum. The MPL allowed for normal executions is 1, no matter if the jobs are long or short. But in the case of being applied to the backfilled jobs, it is 4. This number is justified in chapter 4, when it was configured the virtual malleability mechanism.

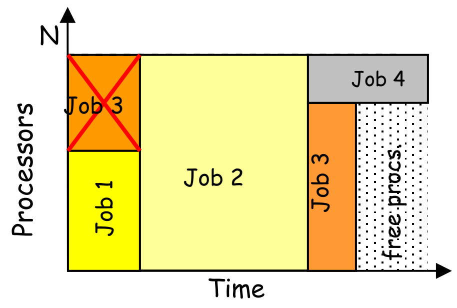
This dynamic modification of the size of the partitions of the backfilled jobs, allows that if later there were resources available, they could modified again. In other works, the previously backfilled and shrunk jobs could be able to expand.



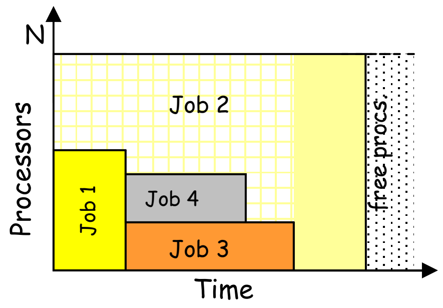
(a) FCFS



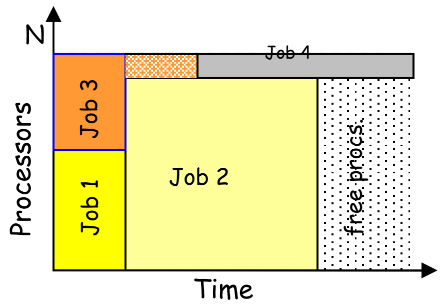
(b) Aggressive backfilling



(c) Moldability + aggressive backfilling



(d) FJT



(e) FJT + backfilling with virtual malleability

Figure 6.2 Job scheduling alternatives.

Figure 6.2 shows examples of different alternatives for job scheduling. It is also compared the proposal of this chapter (d). All the jobs arrive at the same time, where job1, job3 and job4 are short and job2 is long. Job1 cannot be assigned a number of processes greater than N/2 processors and job2 cannot be assigned a number of processes lesser or equal than N/2.

The *FCFS* job schedulilng policy is shown in Figure 6.2 (a). In this case the jobs are treated as rigid ones. Fragmentation was generated because job2 could not start execution until job1 finished, as there weren't enough free resources for it. In addition when job4 finished execution, job3 was not able to take advantage of the new resources available because it was a rigid job.

The jobs in Figure 6.2 (b) are rigid, and aggressive backfilling was applied to job4 to alleviate the fragmentation. But, the execution window for job4 expired so it was aborted and reinserted in the queue to be executed later. An attempt to reduce fragmentation was done, but finally the resources were wasted due to an underestimation of the execution time of job4.

The Figure 6.2 (c) shows an example similar to (b), but in this case the jobs were also moldable. The difference is that in this case it was possible to backfill job3, because it could adapt to the gap generated by job1. As happened in (b) it had to be aborted.

The jobs in Figure 6.2 (d) are treated as moldable, and it is applied the algorithm *FJT* for job scheduling. The fragmentation is completely eliminated, as the jobs were able adapt to the available resources during the whole execution of the workload. However notice that job2 executed totally expanded just at the end of its execution. Moreover, before that it suffered several changes in its partition. On the other hand, the waiting times were reduced.

Finally in Figure 6.2 (e) there is an example of the proposal of this chapter. *FJT* is applied with a maximum MPL equal to 1, which means that jobs are treated as moldable. Job3 is backfilled as in (c), but when the window expired instead of aborting it, virtual malleability is applied to it. In this way, the partition for job3 was reduced and as job2 was moldable it could start immediately. Notice that in this case the partition for job2 is smaller than in the rest of the examples so from the point of view of the application its performance degraded a little. However the machine utilization of the system was improved.

Figure 6.3 shows the mechanism that implements the algorithm *FJT* plus the backfilling techniques with virtual malleability applied to expired windows.

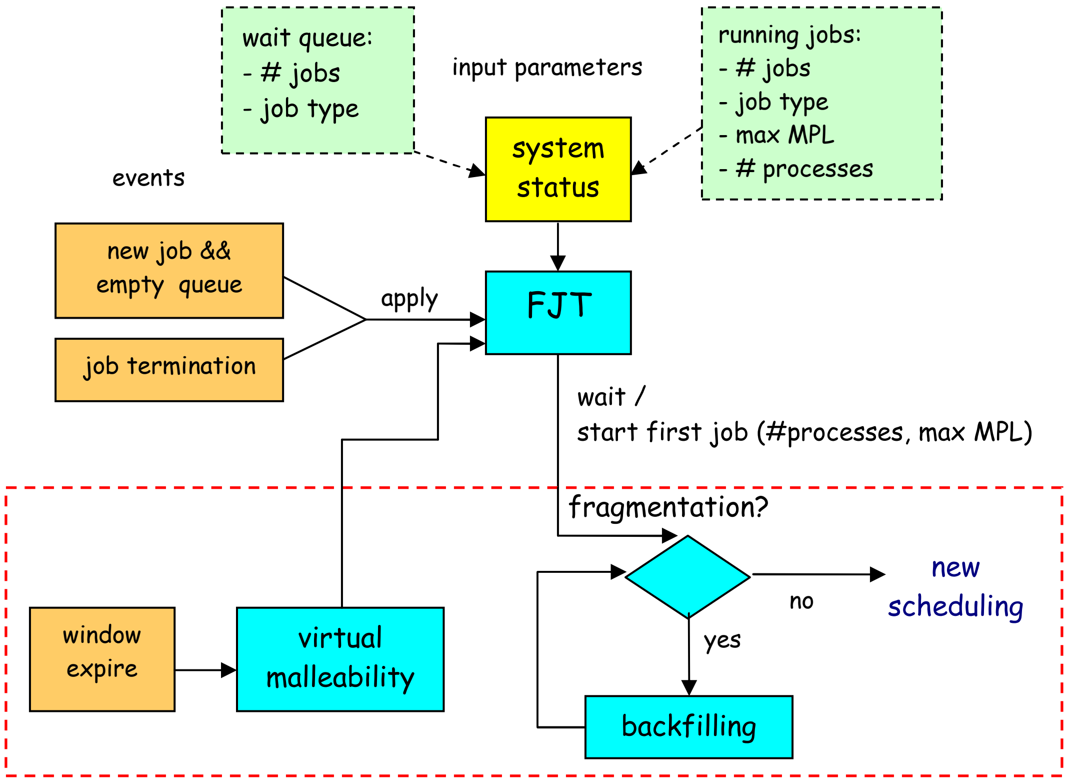


Figure 6.3 Mechanism to implement the algorithm FJT with backfilling and virtual malleability for the expired windows.

The events that trigger the job scheduling algorithm *FJT* are job termination or when the wait queue goes from empty to not empty. The algorithm obtains information from the system like the the number of jobs in the wait queue and currently running, their type (*long* or *short*) and the maximum MPL applied to each of the currently running jobs. The scheme is similar to the one shown in Figure 6.3. The difference is the addition of the possibility of backfilling and the application of virtual malleability to backfilled jobs.

The *launcher* tries to dispatch the job that is at head of the wait queue. If there weren't enough resources for it, its execution is delayed. The launcher makes an attempt to free resources by checking if there are windows expired in which case virtual malleability is applied. If the available resources are not still enough for the first job in the wait queue, the launcher proceeds to apply backfilling.

6.5 Evaluations

This section is dedicated to describe in detail the performance experiments done to evaluate the proposal of this chapter.

6.5.1 Evaluated policies

Next the job scheduling policies used in the evaluations are listed:

a) *FJT* (4): Algorithm FJT proposed in chapter 5. It applies virtual malleability to long jobs and moldability to short jobs. The jobs are dispatched in the same order they arrive.

b) *FJT (1) + backf (abort*): Algorithm FJT proposed in chapter 5, but applying moldability to long and short jobs. It is enhanced with aggressive backfilling. When a window from a backfilled job expires, it is aborted and reinserted in the wait queue.

c*) FJT (1) + backf* (*malleability*): Algorithm FJT proposed in chapter 5, but applying moldability to long and short jobs. It is enhanced with the backfilling techniques, applying virtual malleability to the backfilled jobs which window has expired.

d) *FCFS + backf (abort)* : First-com-first-served classical algorithm with aggressive backfilling. The jobs are treated as rigid and the number of processes chosen for each one is the maximum they can have in a set of 60 processors.

Aggressive and conservative backfilling are the most traditional backfilling techniques, which can be found in production systems. It was chosen aggressive backfilling because in [WeFe01] they demonstrate that when working with moldable jobs it obtained better performance.

6.5.2 Performance results

This section is dedicated to discuss the performance results obtained when evaluating the policies listed in the last section.

The traces for the workloads were generated using the equation described in chapter 3 and were dimensioned in such a way that the last job is launched for execution after 900 seconds of the starting of the first one. Given that the total number of jobs in each workload is approximately 120. The workloads were run on a pool of 60 processors, leaving 4 processos for the CPUM, the launcher and performance analyzing tools [PARA01].

The workloads are composed by applications with different execution times and communication degree. There were considered two family workloads depending on their communication degree which can be seen in Table 6.1 and in Table 6.2. The number of applications is such that the proportion of machine utilization for each one within a given class (*long* or *short*) is equitative. For example, for a high communication degree workload with machine utilization of 20% for short applications, the number of bt.w it is such that they take 10% of the total machine utilization.

Table 6.1 High communication degree workload

workload	**long**	**short**
applications	bt.A, cg.B	bt.W, sweep3D
communication degree	high, high	high, high

Table 6.2 Low communication degree workload

workload	**long**	**short**
applications	ep.C	ep.B, sweep3D
communication degree	low	high, high

As can be seen in Table 6.3, the workloads were designed to have machine utilizations 40, 60, 80 and 100% where 20% is spent by short jobs and the rest is spent by long ones. For more details about how is calculated the machine utilizations approximations refer to chapter 3.

The distribution corresponding to machine utilization of 100% was used in the evaluations in [ZFMS00], which were extracted from a real production system. In addition in the collection of workloads logs available from Feitelson's archive in [Feit06] the distribution percentages between long and short jobs are mostly around 20 to 30% for short jobs and 70 to 80% for long ones.

Table 6.3 Long and short jobs relation within a workload

machine utilization	**100**	**80**	**60**	**40**
% long jobs	80	60	40	20
% short jobs	20	20	20	20

It is important to notice that the number of long jobs represent at most 30% of the total number of jobs.

The results are presented separately for the two categories of jobs: long and short. The response time is calculated as the sum of the waiting time and the execution time grouped by type of job (long or short) and machine utilization.

For each category it is shown the average response time, average wait time and the average execution time.

6.5.2.1 Response time

The average response times expressed in seconds of the worloads mentioned evaluated under the policies listed in section 6.5.1 are presented graphically in the figures below. On the x axis are represented the different machine utilizations and on the y axis are the response time in seconds.

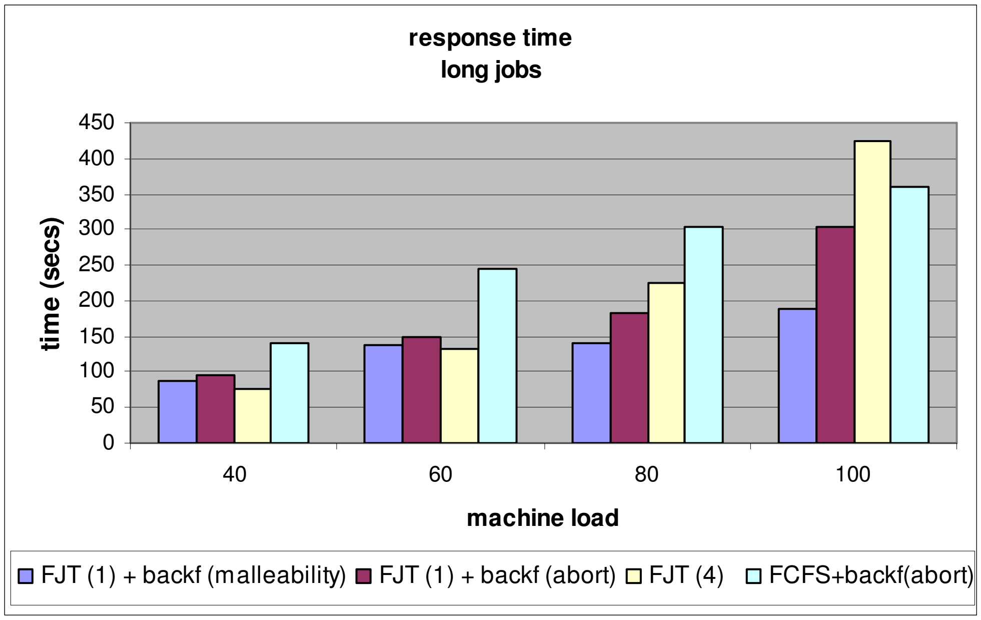


Figure 6.4 Average response times for long jobs in a high communication degree workload

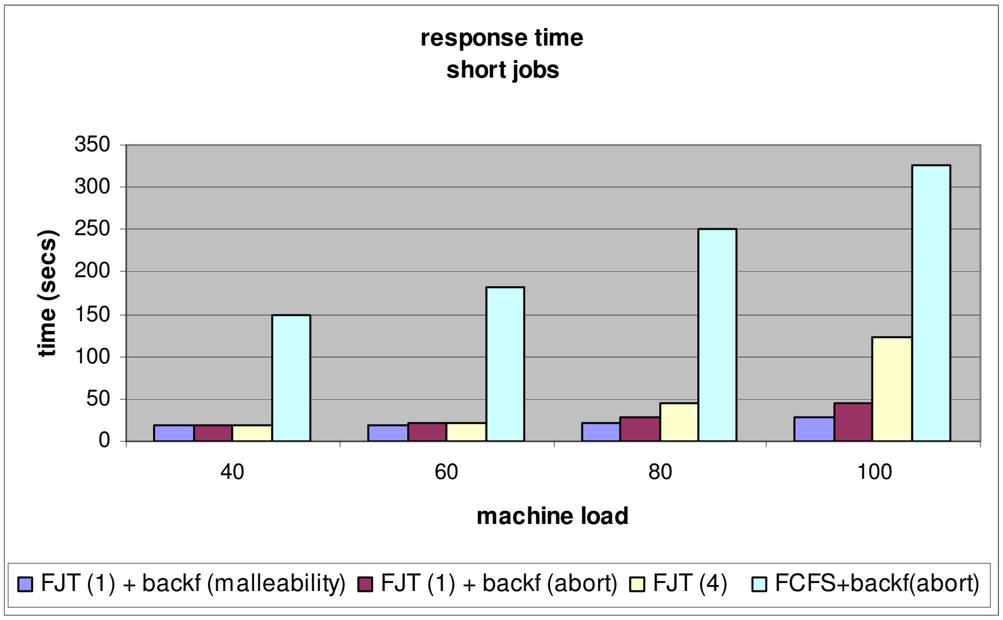


Figure 6.5 Average response times for long jobs in a high communication degree workload

The response times for long and short jobs in high communication degree workloads are shown in Figure 6.4 and Figure 6.5 respectively. As can be observed for low machine utilization, FJT(4) obtained the best performance for long jobs. This is the case when virtual mallealibity is applied to long jobs and moldability to short jobs. For short jobs the performance was similar under any of the policies evaluated.

As the load increments, the performance of FJT (4) is degraded, as well as of FJT(1)+back(abort). However this last one is not as bad as the first one. The difference is greater for short jobs. FJT(1)+backf(malleability) obtained the best performance for high loaded machines.

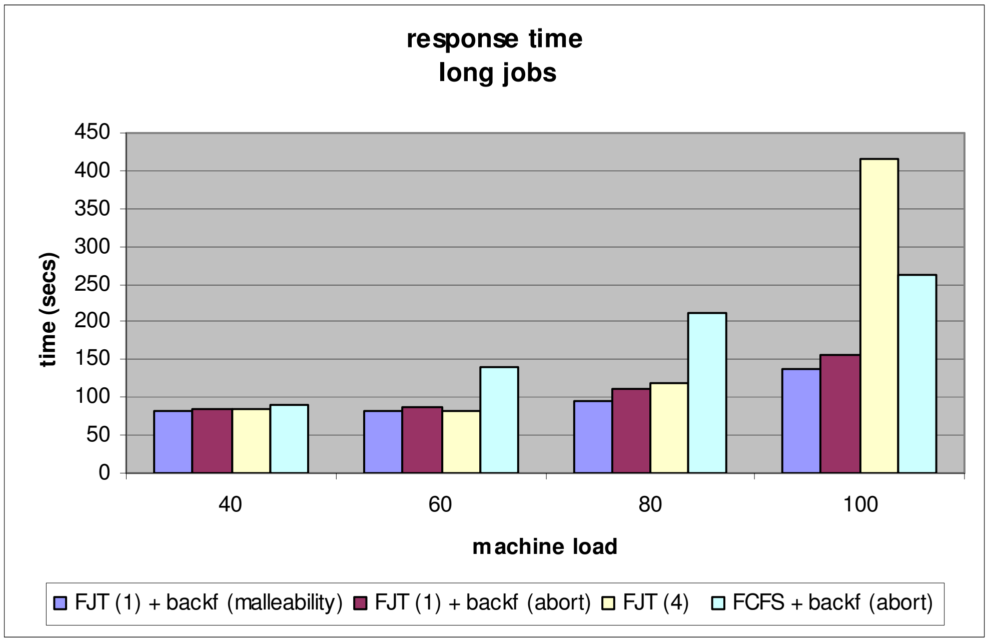


Figure 6.6 Average response times for long jobs in a low communication degree workload

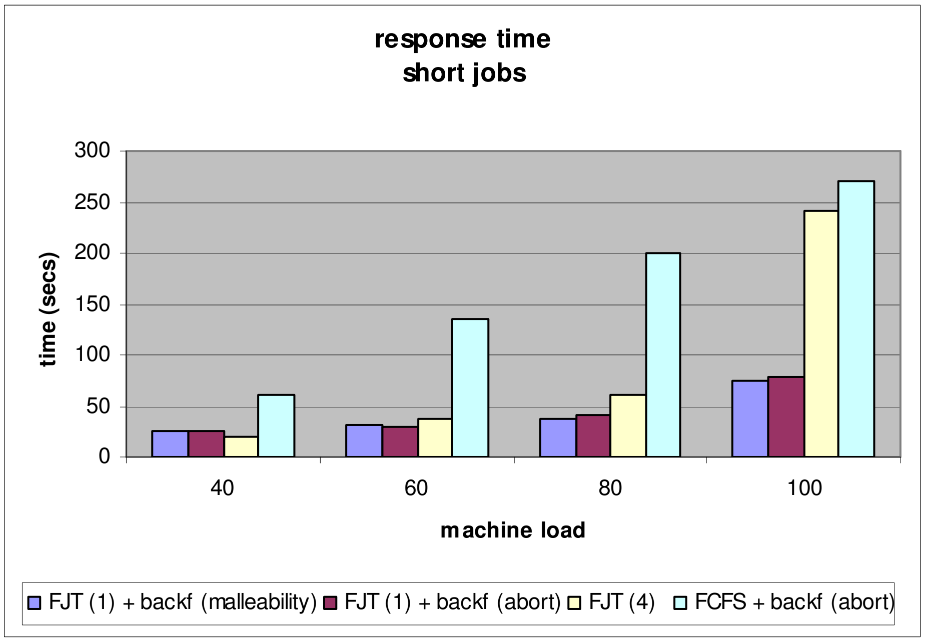


Figure 6.7 Average response times for short jobs in a low communication degree workload

The average response times for long and short jobs in low communication degree workloads are shown in Figure 6.6 and Figure 6.7 respectively. The performance of the evaluated policies was similar for low machine utilization for long and for short jobs. For high machine utilization it can be observed that FJT(4) degraded dramatically. FJT(1)+backf(malleability) obtained the best performance followed by FJT(1)+backf(abort).

6.5.2.2 Average wait and execution time

The average wait times for long and short jobs in high communication degree workloads are shown in Figure 6.8 and Figure 6.9 respectively. The average wait times for long and short jobs in low communication degree workloads are shown in Figure 6.10 and Figure 6.11 respectively.

It can be observed that the wait time for FJT(1)+backf(malleability) was constant for long and for short jobs. For the rest of the policies evaluated the wait time increased as the machine load increased.

As can be expected FJT(4) for long jobs and low machine utilizations had the smallest wait time. This is due to the fact that it allows the jobs to adapt easily to the available resources and in this way they are able start execution immediately, usually shrunk. However as the load increments, this is not a good strategy anymore. The applications are not able to expand so their execution time increases dramatically degrading performance, and consequently the average wait time.

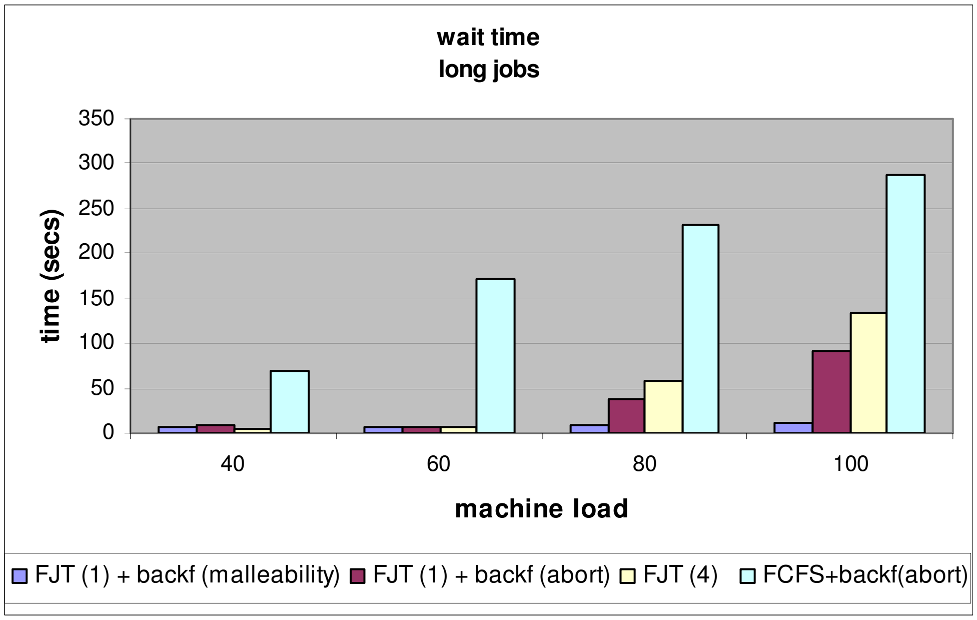


Figure 6.8 Average wait times for long jobs in a high communication degree workload

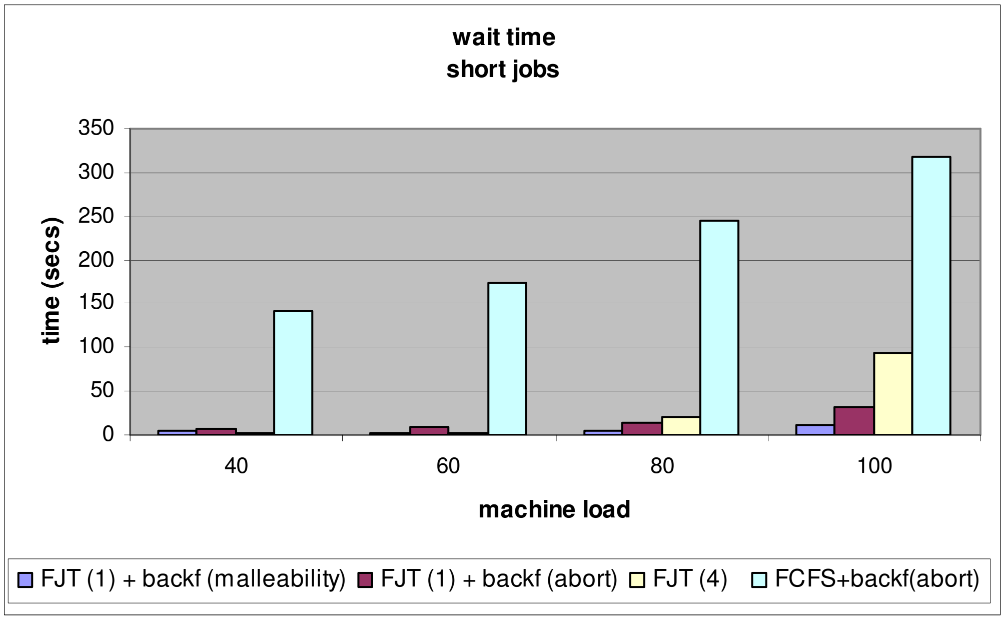


Figure 6.9 Average wait times for short jobs in a high communication degree workload

In FJT+backf(abort) the element that increase the response time in short jobs is the wait time. This last is due to the policy applied when the windows expires. The backfileld jobs are aborted and reinserted in the wait queue. While in the FJT(1)+backf(malleability) even the size of their partition is reduced, they can continue their execution. This penalization in the execution time can be seen in Figure 6.12.

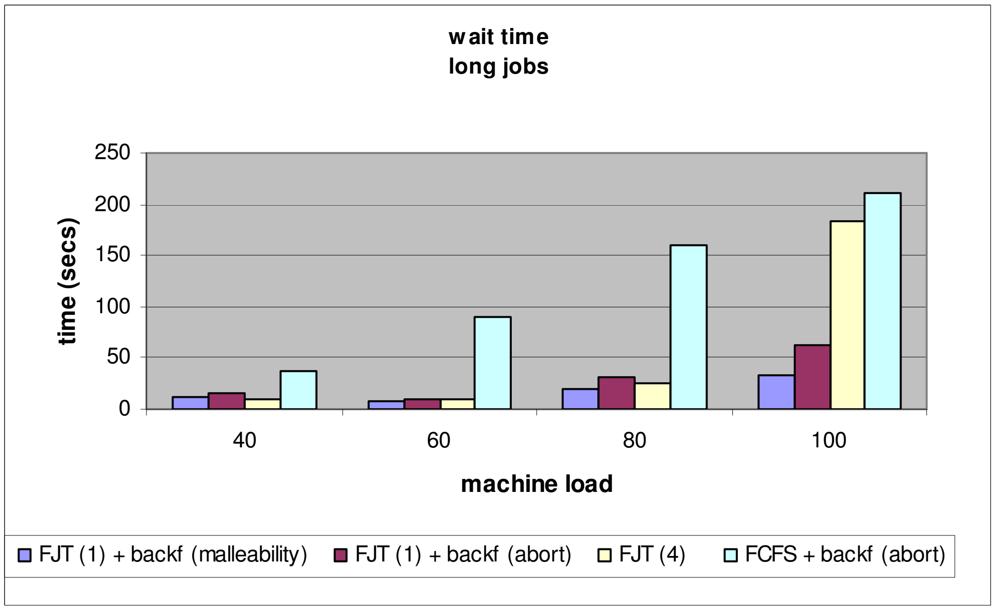


Figure 6.10 Average wait times for long jobs in a low communication degree workload

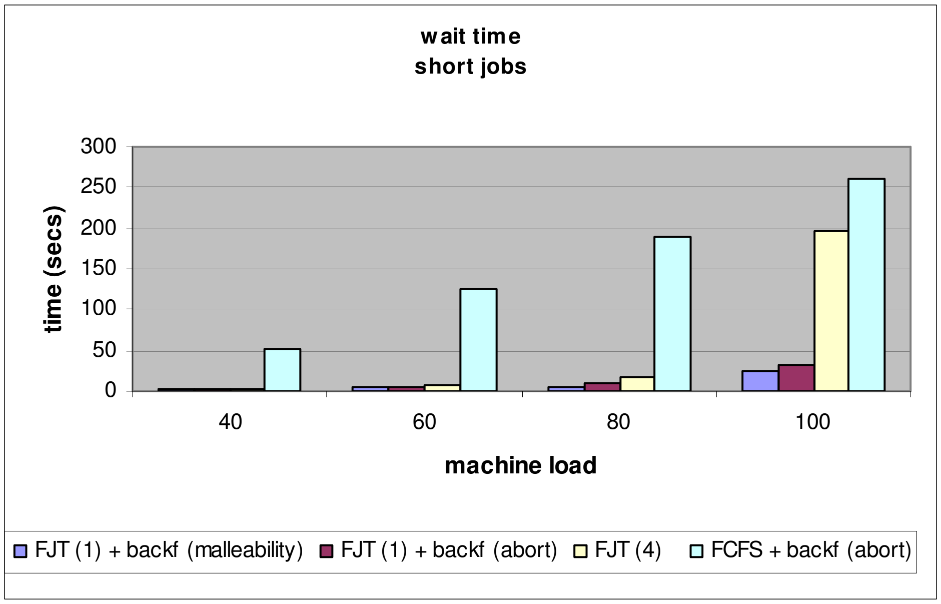


Figure 6.11 Average wait times for short jobs in a low communication degree workload

The average execution times for long and short jobs in high communication degree workloads are shown in Figure 6.12 and Figure 6.13 respectively. The average execution times for long and short jobs in low communication degree workloads are shown in Figure 6.14 and Figure 6.15 respectively.

The element that increased the response times in FJT(4) was the execution time. When the jobs are long this is more evident as they have to be executed shrunk, mainly with high load.

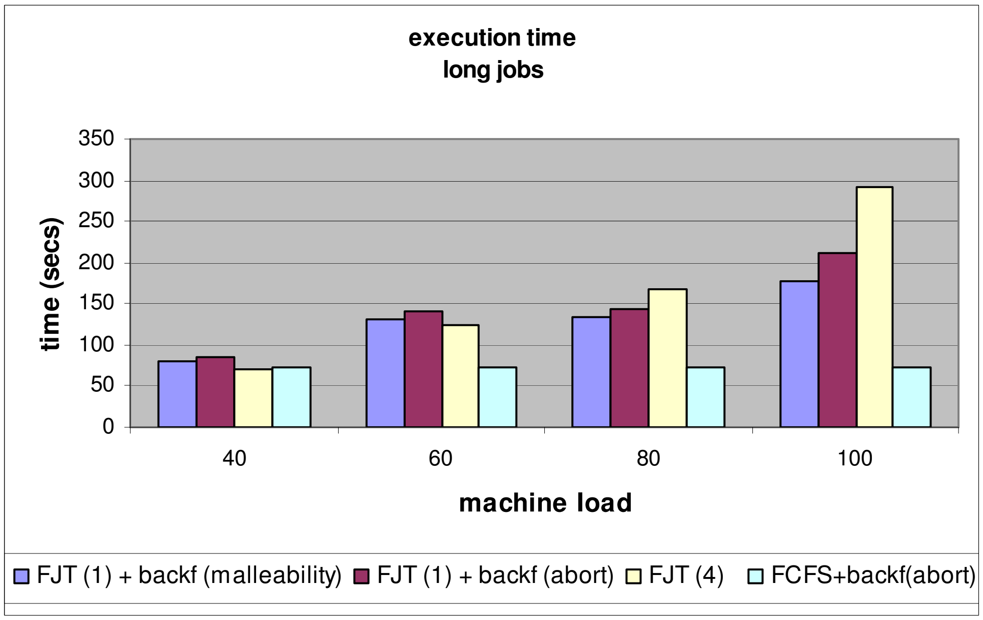


Figure 6.12 Average execution times for long jobs in a high communication degree workload

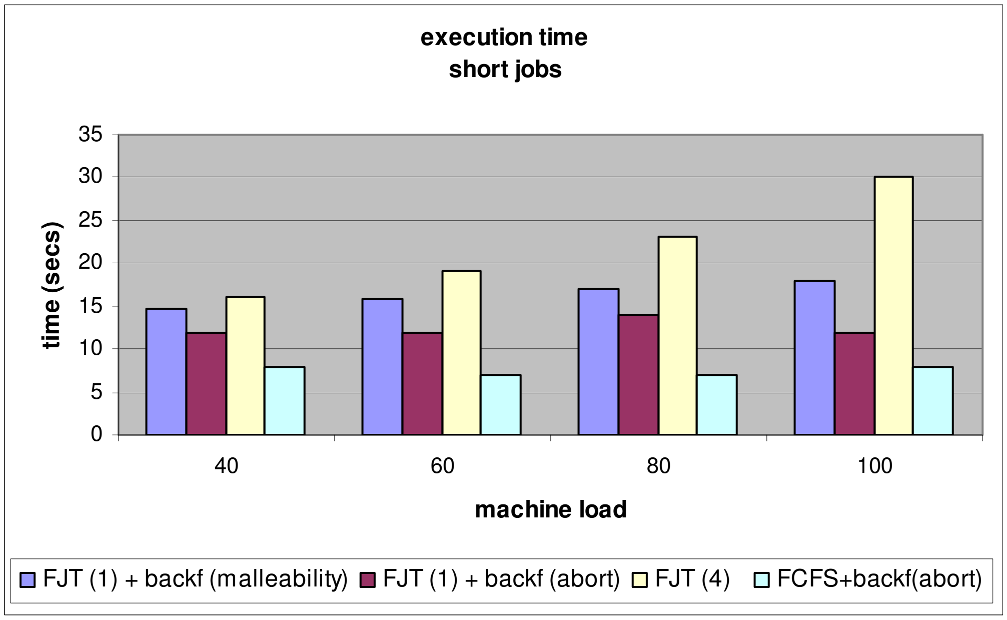


Figure 6.13 Average execution times for short jobs in a high communication degree workload

Another observation is that the relative performance of the evaluated policies in high communication degree workloads is similar to low communication degree workloads. The main impact is given by the machine utilization and the percentage of long and short jobs. This is because except for FJT(4), the rest of the policies evaluated run in isolation, their MPL is equal to 1, so they don't have synchronization problems.

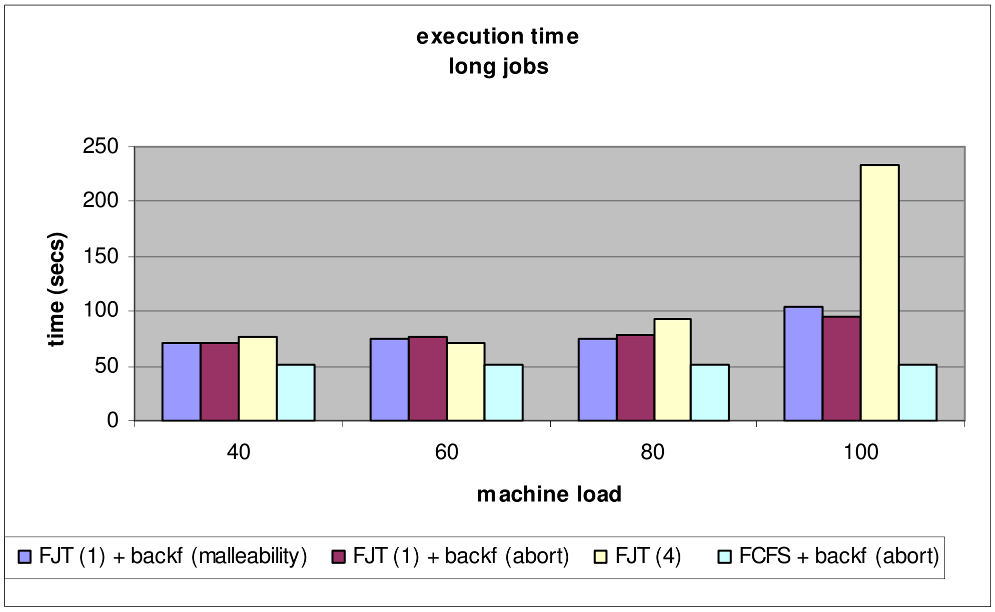


Figure 6.14 Average execution times for long jobs in a low communication degree workload

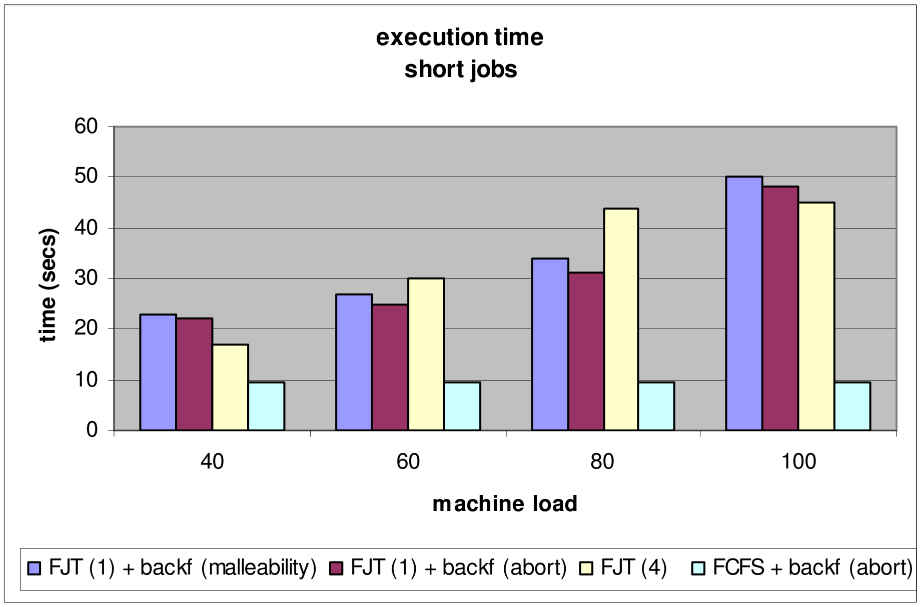


Figure 6.15 Average execution times for short jobs in a low communication degree workload

In conclusion for low machine utilization the performance between all the policies evaluated was very similar, being FJT(4) the one with the smallest response time. As the load increments FJT(4) degraded and FJT(1)+backf(malleability) obtained the best performance.

It was also observed when detaching the response time in execution and wait times that the benefit obtained in the proposal of this chapter FJT with backflling and virtual malleability for the expired windows came from the reduction in the wait times.

Under this policy, it was not dramatically for short jobs when their window expired as they could continue execution and the resources were not wasted as happened with FJT(1)+backf(abort).

6.6 Summary

This chapter proposed adding backfilling to the job scheduling algorithm *FJT*. In addition, a new alternative for the backfilled jobs when they expire their window time is proposed, which consists of applying virtual malleability to them.

FJT algorithm is in charge of setting parameters like the number of processes and the maximum multiprogramming level (MPL) of each application before executing it, as well as when to execute it. The decisions are based on information about the characteristics (i.e. number and type) of the jobs currently running and in the wait queue. This algorithm showed to be too optimistic in a context of high machine utilization. In addition, the overhead generated when stressing applications to be shrunk and expanded so often is very application depended, which needs to be analyzed separately. So, MPL was forced to be set to 1 (moldability) when working with such machine loads.

Backfilling techniques were added to alleviate fragmentation generated when incrementing the load, because of moldability and application characteristics (not all the applications accept any number of processes).

The effectiveness of backfilling depends on user time estimations, which leads to underestimations and overestimations. It had been proposed an alternative in case of underestimation, that is when the window time of a backfilled job expired. In the classical approximations, that is to say conservative and aggressive backfilling, the backfilled jobs are aborted and reinserted in the wait queue. The alternative proposed is to apply virtual malleability to them. In this way the jobs are not aborted and the work done was not wasted. Even their partition is reduced; they could continue execution without stopping and freeing resources for the job at the head of the queue.

All the combinations were implemented, evaluated and compared. The performance results showed that *FJT+backfilling and virtual malleability* obtained the best performance, especially with high machine utilizations.

Chapter 7
CONCLUSIONS AND FUTURE WORK

7.1 Introduction

This dissertation was centered on improving the performance in terms of response time and machine utilization when scheduling parallel applications that communicate via message passing. The platform object of this study was a shared-memory multiprocessor which is actually the backbone of clusters of SMPs.

To achieve that objective, this work has analyzed at job, processor and process level several scheduling strategies taking advantage of the flexibility a job could offer and the knowledge of the system.

MPI was the message passing library used, for being the most widely used and for its portability across shared and distributed memory architectures.

The evaluations were done on real executions. In order to do that it was developed an execution environment composed by a queuing system named *Launcher*, which was in charge of the job scheduling, a resource manager named *CPU Manager*, which was in charge of processor allocation and a *VMruntime library* which was in charge of doing the process scheduling. These components communicate and coordinate each other via shared memory.

Next the conclusions for each of the contributions of this work are described.

7.2 Conclusions

Virtual malleability was developed with the aim of making applications able to adapt to load changes. During execution applications can vary the size of their processor partition by incrementing or decrementing their multiprogramming level. The number of processes is set at the beginning of the execution and it remains fixed for the whole execution.

At system level, the *virtual malleability* mechanism can be exploited by setting parameters such as number of processes, maximum multiprogramming level and starting execution time. In this dissertation an algorithm named *Folding by JobType* (*FJT*) was proposed to set such parameters based on information taken from the actual system state with the objective of reducing response time and fragmentation. This algorithm forms part of the *launcher*.

FJT takes advantage of variations of the machine load, especially when applications are scheduled during high peaks. However, the algorithm takes optimistics decisions which could not result in good performance when working with heavy loaded machines. In this case, it is applied just moldability and fragmentation is alleviated by adding *backfilling* techniques to the algorithm.

Finally it was proposed a new alternative to the backfilled jobs when their execution time is underestimated. Anytime their window time expires, instead of aborting or suspending the backfilled jobs, virtual malleability is applied to them.

In the following section are described the benefits and drawbacks of the proposals of this work.

7.2.1 Virtual Malleability

This mechanism was developed to modify at runtime the partition size assigned to a job with the objective of improving performance and reducing fragmentation.

It was demonstrated that an application when competing for the resources with itself than with other applications, obtained better performance, especially in workloads with high machine utilization. Concerning the synchronization problem, it was observed that blocking immediately when there is no useful work to do was the best option for processes from high communication degree parallel jobs.

It was proposed a mechanism, the *load balancing detector* (LBD) [UtCL0905], to classify applications dynamically in well-balanced or imbalanced, without any previous knowledge of them, and apply local or global queue to each job.

The *LBD* applies to each job independently, that means that in a workload there may be jobs executing with different balance degree and consequently applying different queue types.

7.2.2 Folding by Job type

It was proposed an algorithm, *Folding by Job Type* (*FJT*), which decides when to launch each job from the wait queue, and at the beginning of execution determines the number of processes, as well as the maximum multiprogramming level.

In general, *FJT* obtained better performance than the rest of the techniques evaluated under workloads with a high coefficient variation of arrivals. This proposal got benefit especially from workloads with bursty arrivals.

7.2.3 Folding by Job type with backfilling

It was added backfilling to the *FJT* algorithm. And it was proposed a new alternative to improve traditional backfilling when the execution times of the jobs were underestimated. That is, when the window execution time of a backfilled job expires and it hasn't finished execution yet. Instead of aborting or suspending and reinserting the job in the queue, the proposal is to reduce its partition size by applying *virtual malleability*. As moldability is allowed, it increases the possibility of finding a suitable partition to backfill a job or to enter executing.

The complete proposal was implemented, evaluated and compared with other moldability and backfilling techniques under several dynamic workloads. It demonstrated to outperform the rest of the evaluated policies by about 20 to 30% especially when executing on high loaded machines.

It reduces memory swapping generated by aborts/suspensions, prevents the queuing system from reinserting in the queue and re-executing the job in the future. Notice that if the job is reinserted in the queue it will be eligible again to be backfilled.

7.3 FUTURE WORK

The evaluated and proposed policies are currently implemented to run on one node. The idea is to port them to a cluster of SMPs. This could be done with some restrictions, concerned with running processes all of the same application or at least the sub-group of processes that are to be applied *virtual malleability*, on the same node. Then in a node it will be possible to apply *virtual malleability* without extra costs, as in a SMP.

For the moment all the process scheduling have been done through a runtime library. This library performs the scheduling by an interposition mechanism. It would be interesting to incorporate the implementation of this scheduling level inside the message passing library gaining more control and flexibility over the mechanism. Furthermore, it could be added to other programming models such as *UPC*.

The performance for shrunk applications can be improved if the mapping were done in a more complex way, taking into account the internal communication pattern of the application as well as its balance degree.

As the evaluations taken in this work were based on real executions, there existed practical limitations on the duration and composition of the workloads. In order to extend the experiments to a wider range of workloads, it could be interesting to implement and evaluate the contributions on a simulator.

The efficiency of the *FJT* algorithm depends mostly on the accuracy of the “predictions” to set the parameters when dispatching the jobs. The overhead generated when an application is shrunk and expanded is not the same for all the applications. A mechanism that could learn from it particular application could help to improve the accuracy and minimize the overhead.

Finally, another future objective is to apply virtual malleability to other situations where preemption is involved as a solution, trying to bring the possibility of malleability as an alternative. Real time systems are an example of such systems.

BIBLIOGRAPHY

[AnLL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy, The performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors, IEEE Trans. on Comp., 38(12):1631-1644, Dec. 1989

[ArCu01] A. C. Arpaci-Dusseau, D. Culler. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. ACM Trans. Compu. Sys. 19(3), pp.283-331, Aug. 2001.

[AtSe88] W.C.Athas, C.L.Seitz. Multicomputers: message-passing concurrent computers. Computer 21(8), pp. 9-24, Aug 1988.

[BaKi92] Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint.

[BCAY06] Christopher Barton, Calin Cascaval, George Almasi, Yili Zheng, Montse Farreras, Siddhartha Chatterjee and Jose Nelson Amaral. Shared Memory Programming for Large Scale Machines. ACM SIGPLAN Conference on Programming Language Design and Implementation. Journal PLSI 2006.

[BeLL89] B.N. Bershad , E.D. Lazowska, H.M.Levy , The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors, IEEE Trans. on Comp., 38(12):1631-1644, Dec. 1989.

[Berk06] http://upc.lbl.gov/

[Berl02] K. Berlin. Draft: UPC vs MPI and OpenMP: Analysis of a Hybrid Approach to Parallel Programming. 2002.

[BHJK03] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan. Evaluating the Impact of Programming Language Features on . the Performance of Parallel Applications on Cluster Architectures. LNCS 2958, pp 194-208. 2004.

[BHSW95] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA, December 1995.

[BKSH01] M. Bhandarkar, L. V. Kale, E. de Sturler,and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In Proc. of the Int. Conf. on Comp. Science, San Fr., CA, LNCS 2074, pp 108–117, May 2001.

[Blac00]D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. Computer 23(5), pp. 35-43, May 1990. [16] Silicon Graphics, Inc. IRIX Admin: Resource Administration, Document number 007-3700-005, http://techpubs.sgi.com, 2000.

[Blac90]D. L. Black, Scheduling support for concurrency and parallelism in the Mach operating system. Computer 23(5), pp. 35-43, May 1990.

[BRCR91] R. M. Bryant, H-Y Chang, and B. Rosenburg, Experience developing the RP3 operating system. Computing Systems 4(3), pp. 183-216, Summer 1991.

[CaZa94] C. McCann, J. Zahorjan. Processor allocation policies for message passing parallel computers. In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 19-32, May 1994.

[Cirn01] W. Cirne. Using Moldability to Improve the Performance of Supercomputer Jobs. Ph.D Work. Computer Science and Eng. University of California San Diego, 2001.

[COML00] J. Corbalan, X. Martorell, J. Labarta. Performance-Driven Processor Allocation. Proc. of the 4th Operating System Design and Implementation (OSDI 2000), San Diego, CA, October 2000.

[DeIy89] M. V. Devarakonda, R. Iyer. Predictability of Process Resource Usage: A Measurement Based Study on UNIX. IEEE Trans. Soft. Eng. 15(12), pp.1579-1586, Dec. 1989.

[DHHW93] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[Down97] A. Downey. A Model for Speedup of Parallel Programs. Technical Report CSD-97-933. University of California at Berkerley,1997.

[DUAC96] A.C. Dusseau, R.H. Arpaci, and D.E. Culler. Effective Distributed Scheduling of Parallel Workloads. In Proceedings of the 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA, May 1996.

[DUCM98] A.C.Arpaci-Dusseau, D. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Madison, WI, June 1998

[Feit06] D. G. Feitelson. Logs of real parallel workloads from production systems http://www.cs.huji.ac.il/labs/parallel/workload/swf.html

[Feit97] D.G.Feitelson and M.A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. Job Scheduling Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science. Springer-Verlag 1997.

[FeJe97]D.G.Feitelson and M.A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D.G.Feitelson and Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science. Springer-Verlag 1997.

[FeNi95] D. G. Feitelson, B. Nitzberg. Jobs Characteristics of a Production Parallel Scientific Workload on the NASA Ames Ipsc/860. In JSSPP Springer-Verlag, Lectures Notres in Computer Science, vol. 949, pp. 337-360, 1995

[FERU97] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and K. C. Sevcik. Theory and Practice in Parallel Job Scheduling. Lecture Notes in Computer Science, 1291:1--34, 1997

[FFFP03] E. Frachtenberg, D. Feitelson, J. Fernández, F. Petrini. Parallel Job Scheduling Under Dynamic Workloads. In JSSPP 2003.

[FFPF03] E. Frachtenberg, D. Feitelson, F. Petrini, J. Fernández, Flexible CoSheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. In IPDPS'03.

[FPFC02] E. Frachtenberg, F. Petrini, J. Fernandez, S. Coll. Scalable resource management in high performance computers. Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02) 2002.

[FRSS97] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and K. C. Sevcik. Theory and Practice in Parallel Job Scheduling. Lecture Notes in Computer Science, 1291:1--34, 1997.

[GeKP96] G.A.Geist, J.A.Kohl, P.M.Papadopoulos. PVM and MPI: a Comparison of Features.www.csm.ornl.gov/pvm/PVMvsMPI.ps. 1996.

[GrLu02] W. Gropp, E. Lusk. Goals Guiding Design: PVM and MPI. IEEE International Conference on Cluster Computing (CLUSTER'02) p.257-265. 2002.

[GrNR90] D.C.Grunwald, B.A.A.Nazief, D.A.Reed. Empirical comparison of heuristic load distribution in point to point multicomputer networks. In 5th Distributed Memory Computing Conference, pp. 984-993, 1990.

[Gupt89] R.Gupta. Synchronization and Comunication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems. In ICPP'89. pp II:23-30,1989.

[GuTU91] A.Gupta, A.Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Jobs. In Proceedings of the 1991 ACM SIGMETRICS Conference, pages 120-132, May 1991.

[HDGH93] F. Hofmann, M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, C¬ Linster, T. Thiel, S. Turowski. MEMSY: a modular expandable multiprocessor system. In Parallel Com¬p. Arch., A. Bode and M. Dal Cin (eds.), pp. 15-30, Springer Verlag, 1993. LNCS Vol. 732.

[hpcupc00] http://h30097.www3.hp.com/upc/

[IBMU] http://www.alphaworks.ibm.com/tech/upccompiler/download

[IBSP03] M. Islam, P. Balaji, P. Sadayappan, D.K.Panda. QoPS: A QoS based scheme for Parallel Job Scheduling. LNCS 2862 pp 252-268. 2003.

[IBSP04] M. Islam, P. Balaji, P. Sadayappan, D.K.Panda. Towards provision of quality of service guarantees in job scheduling. IEEE International Conference on Cluster Computing, 2004.

[ISOC99] http://www.open-std.org/jtc1/sc22/wg14/www/docs/C99RationaleV5.10.pdf

[JaSC01] D. Jackson, Q. Snell and M. Clement. Core Algorithms of the Maui Scheduler. In Worshop on Job Scheduling Strategies for Parallel Processing, pp. 87-102, 2001.

[KlPa84] D. Klappholz, H-C Park. Parallelized process scheduling for a tightly-coupled MIMD machine. In Intl. Conf. Parallel Processsing, pp. 315-321, Aug. 1984.

[lanmpi] http://www.lam-mpi.org/

[LaSm02] B. Lawson and E. Smirni. Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems. In Job Sched. Strategies for Parallel Processing, D.G. Feitelson and L. Rudolph (eds.), Springer Verlag, Lect. Notes Comp. Sc. Vol. 2537, pp. 72-87, 2002.

[LeSB88] T. LeBlanc, M. Scott, C. Brown. Large¬scale parallel programming: experience with the BBN Butterfly parallel processor. In Proc. ACM/SIGPLAN, pp. 161-172, Jul. 1988.

[Lifk95] D.Lifka. The ANL/IBM SP scheduling system. In Job Scheduling Strategies for Parallel Processing, pp. 295-303, Springer Verlag, 1995 (LNCS 949).

[LiHu89] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems, vol. 7, pp. 321-359, Nov. 1989.

[MCFF98] J.E.Moreira , W. Chan, L.L.Fong, H.Franke, M.A.Jette. An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. In Supecomputing'98, Nov. 1998.

[MCNN00] X. Martorell, J. Corbalán, D. Nikolopoulos , J. I. Navarro , E. Polychronopoulos , T. Papatheodorou , J. Labarta. A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager. LNCS, 1911, pp 55-69. Springer 2000.

[mpich] http://www-unix.mcs.anl.gov/mpi/mpich2/

[MPI94] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. Int. Journal of SuperComputer Jobs, 8(3/4):165-414, 1994.

[Nas03] www.nas.gov/News/ Techreports/2003/PDF/nas-03-010.pdf

[NBSD99] S.Nagar, A.Banerjee, A.Sivasubramaniam, and C.R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In Eleventh ACM Symposium on Parallel Algorithms and Arquitectures, SPAA'99, Saint-Malo, France, June 1999.

[OpenMP05] www.openmp.org. OpenMP Forum.

[openmpi05] http://www.open-mpi.org/papers/lanl-2005-red-storm/

[PaDo96] J. D. Padhye and L. Dowdy, "Dynamic versus adaptive processor allocation policies for message passing parallel computers: an empirical comparison". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), pp. 224--243, Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[PARA01] Parallel Program Visualization and Analysis Tool. 2001. http://www.cepba.upc.edu/paraver

[Pier88] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384--390. ACM Press, 1988.

[RSSD95] E. Rosti, E. Smirni, G. Serazzi, L. W. Dowdy. Analysis of Non-Work-Conserving Processor Partitioning Policies. In JSSPP 1995, 165-181 and Arquitectures, SPAA'99, Saint-Malo, France, June 1999.

[Sark89] V. Sarkar. Determining Average Program Execution Times and Their Variance. In Proc. SIGPLAN Conf. Prog. Lang. Dessign and Implementation, pp. 298-312, Jun 1989.

[Se00NC] Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In Proc. USENIX Annual Technical Conf., pp 225--238, 2000.

[ShFe03] E. Shmueli, D. G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.), pp. 228-251, Springer-Verlag, 2003. Lecture Notes in Computer Science Vol. 2862.

[SiGr00] Silicon Graphics, Inc. IRIX Admin: Resource Administration, Document number 007-3700-005, http://techpubs.sgi.com, 2000.

[SKSS02] S. Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling. Proceedings of the 2002 International Conference on Parallel Processing Workshops. pp 514. 2002.

[SLMB90] M. Scott, T. LeBlanc, B.Marsh, T.Becker, C. Dubnicki, E. Markatos, N.Smithline, Implementation issues for the Psyche multiprocessor operating system. Comp. Syst. 3(1), pp. 101-137, 1990.

[SnCJ02] Q. Snell, Mark J. Clement, David B. Jackson: Preemption Based Backfill. JSSPP 2002: 24-37.

[SoPC98] W.E.W. Patrick Sobalvarro, Scott Pakin and A.A.Chien. Dynamic Coscheduling on Workstation Clusters. In D.G. Feitelson and Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1459 of Lecture Notes in Computer Science, pages 231-256. Springer-Verlag, 1998.

[SqNe91] M.S. Squillante and R.D.Nelson , Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling, In Proc. of the 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Comp. Syst., pp 143-145, May 1991.

[SSKH02] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In Proceedings of the 9th Intl. Conference on High Performance Computing, Dec. 2002.

[Sweep] Sweep3D Benchmark http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html

[TaFe99] D. Talb, D. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In 13th Intl. Parallel Proc. Symp. (IPPS), pp.513-517, Apr. 1999.

[ThCr98] R.Thomas, W. Crowther. The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors. In Proc. of the ICPP'88, pp 245-254, Aug. 1998.

[TuGu89] A. Tucker, A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In Proc. of the SOSP'89, pp. 159-166, Dec. 1989.

[upc] http://upc.gwu.edu/

[UtCL03] G. Utrera, J. Corbalán, J. Labarta. Study of MPI applications when sharing resources. In Technical Report number UPC-DAC-2003-47, 2003. http://www.ac.upc.es/recerca/reports/DAC/2003/index,en.html.

[UtCL0904] G. Utrera, J. Corbalán, J. Labarta. Scheduling of MPI applications: Self coscheduling.Euro-Par 2004, Lecture Notes in Computer Science 3149, pp 238-245.

[UtCL1004] G. Utrera, J. Corbalán, J. Labarta. Implementing Malleability on MPI Jobs. In Proceedings of the Parallel Architecture and Compilation Techniques, 13th International Conference on (PACT'04), pp. 215-224, Antibes Juan-les-Pins, France, Sep 29 - Octubre, 2004.

[UtCL0605] G. Utrera, J. Corbalán, J. Labarta. Another approach to backfilled jobs: applying Virtual Malleability to expired windows. In Proceedings of ICS'05 pp 313-322, June 2005-

[UtCL0905] G. Utrera, J. Corbalán, J. Labarta. Dynamic load balancing. In Proceedings of 6th International Symposium on High Performance Computing (ISHPC-VI), September 2005.

[VAZA91] R.Vaswani, J. Zahorjan. Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors,In Proc. SOSP'91 pp 26-40, Oct. 1991.

[WaMa02] W. Ward Jr., C. L. Mahood, J. E. West. Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy. JSSPP 2002.

[WeFe01] A.M Weil and D. Feitelson. Utilization, Predictability, Workloads and User Runtimes Estimates in Scheduling the IBM SP2 with Backfilling. In IEEE Trans. on Parallel and Distributed Syst. 12(6), pp.529-543, Jun. 2001.

[ZFMS00] Y. Zhang, H. Franke, J.Moreira, A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. IPDPS 2000.

[Zeeh04] Christina Zeeh. Overview of the MPI Standard and Implementations. MPI Report. http://tuxtina.de/files/hauptseminar/mpireport.pdf. Cluster computing, May 2004.