Implementing Malleability on MPI Jobs

Gladys Utrera, Julita Corbalán, Jesús Labarta
{gutrera, juli, jesus}@ac.upc.es
Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)

Abstract. Parallel jobs characterized for having processes that communicate and synchronize each other frequently. A processor allocation strategy widely used in parallel supercomputers to optimize their performance is Space-Sharing, that is assigning a processors partition to each job for its exclusive use.

The Moldability can reduce the queuing time by sizing jobs to the available resources at the start of execution. However, they will remain fixed even though the system load varies. Only Malleable jobs will be able to adapt to load changes. A way to simulate Malleabilty using Moldability is by applying the Folding technique.

In this article we propose a processor allocation strategy to message-passing parallel jobs, based on Folding and Moldability concepts. This technique tries to decide the optimal initial number of processes, when to fold and the number of Folding times by analyzing the current and past system information. At processor level, we apply Co-Scheduling.

We implement our proposal and compare to an implementation of Folding and some other Moldability from the bibliography. We evaluate them under several workloads with different job sizes, classes and machine utilization. Results show that the proposed technique adapts easily to load changes, and can obtain better performance than the rest evaluated, on workloads with high coefficient variation and especially with burst arrivals.

1 Introduction

Parallel jobs characterized for having processes that communicate and synchronize each other frequently. A processor allocation strategy widely used in parallel supercomputers to optimize their performance is Space-Sharing, that is assigning a processors partition to each job for its exclusive use.

When a new job arrives to the system it will start execution as soon as there is an available partition that satisfies its requirements, meantime it will stay in the wait-queue.

A job is said to be Moldable when it can executes in different partition sizes. However, once it starts execution, the partition size can't be modified. For this reason the decision over the partition size will determine the overall performance of the execution. A job is said to be Malleable if it has also the possibility to modify the partition size during execution.

The Moldability can reduce queuing time because jobs are sized to the available resources at the beginning of the execution. However, they will remain fixed even though the system load varies. Only Malleable jobs will be able to adapt to load changes.

There is much work about techniques that decides the optimal number of processes to execute a Moldable job [2], [3], [4]. In general they make their decision based on the number of jobs in the wait-queue, the known system load and the recent past system behaviour. A way to simulate Malleabilty using Moldability is by applying the Folding technique [2]. This means to reduce its partition size in a half, duplicating the number of processes per processor partition.

We will focus on message-passing parallel jobs, using the MPI library. These types of jobs aren't Malleable, but can be Moldable.

In this article we propose a processor allocation strategy based on Moldability and Folding techniques, the Folding by JobType (FJT). Our proposal differs from the existing ones in the decisions made for the choice of the initial number of processes, the number of Folding

times, when to fold and at processor level, how processors will be shared, Co-Scheduling instead of pure Time-Sharing.

The proposed technique is implemented and compared to an implementation of the Folding combined to Moldability [3] and some pure Moldabilities techniques, like the ASP and PSA [4]. We evaluate them under workloads with different job sizes and classes and machine utilization. Results show that our proposed technique, the FJT, adapts easily to load changes. In general we obtain better performance than the rest of the techniques evaluated workloads with high coefficient variation, especially with burst arrivals.

The rest of the paper is organized as follows: In Section 2 we present the related work. Then in Section 3 we describe our proposed technique, the FJT. Next in Section 4 we describe the execution framework. After that in Section 5 we describe the experimental platform, the applications and the workloads, and following in Section 6 we show the performance results and finally in Section 7 the conclusions remarks and future work

2 Related Work

In [4] are described and evaluated two Moldability family techniques: Work-Conservative and Non-Work-Conservative. That is if all the processors are assigned to a job's partition or some of them are left idle for future arrivals. They demonstrate that Non-Work-Conservative policies have good performance when system behaviour and load are irregular. This includes: poor job scalability, multi-class workloads, burst arrivals and system failures. They evaluate by simulation the Work-Conservative and Non-Work-Conservative counter part of the ASP-MAX and PSA. In the ASP-MAX, the number of processors assigned to each job must be less or equal to the minimum job available parallelism and a MAX parameter. Under PSA the number of partitions is calculated taking into account the number of queued jobs. If the calculated partition size is greater than the number of free processors, a Non-Work-Conservative decision is made, that is no job begins execution and the processors keep idle. 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 (CV[1]), there is a great variability between arrivals and are by bursts.

In [3] they apply Moldability to a technique proposed in [2] 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 high. However, when load goes down, it is unable to get benefit from the new available processors.

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

On the other hand, at processor level, when jobs are folded, we apply Co-Scheduling techniques as in [8] we showed that for message-passing parallel jobs, these policies can obtain a significant gain over Time-Sharing.

[1] Coefficient Variation of a range of values is the result from dividing the Standard Deviation into the Mean.

3 Proposed Technique: FOLDING BY JOBTYPE (FJT)

As we have already said we will work on jobs that aren't Malleable, just Moldables. The message-passing library used is MPI, from the native operating system, with no modifications. The processes migrations between processors don't need extra synchronization; moreover they are transparent to the user.

We will treat the jobs as belonging to one of two classes: long execution time or short execution time. 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 maximum execution time and other parameters. In other words, each submission queue represents a class of jobs

So given a job, we will assume as known the information about to which class of the mentioned above it belongs to. However, it doesn't mean that it is required to know the exactly execution time which would be unrealistic.

An irregular machine load will have execution peaks. If a job arrives at 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 or start execution with just the few available processors. In the first case, queuing time would become unacceptable and in the second case, it could happen that resources would become available later but the job will be unable to take advantage of them because our jobs aren't Malleable. Moreover if the jobs were one of those Longs, their execution time would increment considerably.

Each time a job arrive to the system it is necessary to take three decisions: 1) executes or wait; 2) if executes folded or not; 3) how many processes.

In order to take these decisions, first the algorithm examines if the arrived job is Long or Short. Below there is a simplified version of the algorithm that decides the described decisions.

```
Arrived Job=Short =>  If there are idle processors or long jobs able to be folded then execute
                      else wait.
Arrived 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 execute folded
                              else if long jobs running unfolded then execute folded
                              else wait
                    else /* there are long jobs queued */
                              if there are idle processors or long jobs running unfolded then
                                        execute unfolded
                              else wait
                    endif
```

The idea is that each time a job arrives to the system, try to deduce from the current context available information, like class of the queued and running jobs, if it is possible that in the near future, more processors would become available. We state that this can happen if one of the situations occurs: 1) there are short jobs running and the MPL=1, this means currently there aren't any jobs running folded; 2) there aren't any long jobs queued. When of the two conditions verifies, in our proposed strategy when a Long Job arrives, will assign to it a number of processes greater than the number of available processors. They will run folded until Short Jobs finish execution. After that Long Jobs will be able to expand to the new 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 aren't idle processors, if there are Long Jobs executing they will be shrinked freeing processors and letting Short Jobs to start execution immediately.

So taking all this into account and applying Folding only to Long Jobs, there are a lot of possible combinations, which we treated separately in order to design our proposal.

We have measured in previous evaluations [7] that the maximum number of Folding times an application can have with an acceptable performance is about 4. So we will apply this number as the maximum when required.

About queued jobs, we distribute the idle processors between them. If there are only Short Jobs queued, we do an equipartition between all the them, as in PSA [3].

The FJT sometimes takes Work-Conservative decisions and sometimes Non-Work-Conservative ones. This means sometimes all idle processors are assigned to the newly arrived jobs, and there are situations when processors are kept idle even though there are queued jobs.

Our objective, apart from reducing queuing time as in Folding, is minimizing the individual execution time by taking advantage of the available resources generated when load varies, augmenting the overall performance.

On the other hand, at processor level, when jobs are folded, we apply Co-Scheduling techniques. In [8] we showed that for message-passing parallel jobs, these policies can obtain a significant gain over Time-Sharing.

4 Execution Framework

In this section we will describe the implementation and characteristics of the queuing system used for the evaluation, the Launcher which implements the processor allocation strategies and the Cpu Manager, which manages the process to processor mapping and the processor scheduling.

4.1 Queuing system: Launcher

The Launcher is the user-level queuing system used in our execution environment. It performs a first come first served policy from a list of jobs belonging to a predefined workload, which is received as a parameter. The Launcher in coordination with the Cpu Manager controls the maximum multiprogramming level (MPL[2]). Once a job arrives to the system it decides its optimal number of processes and the number of Folding times according to the processor allocation policy. After that it launches the job as soon as there are enough processors that satisfy its requirements.

4.2 Cpu Manager (CPUM)

The CPUM is a user-level scheduler. Once the Queuing system has launched the job, it enters under the CPUM control. The CPUM implements the scheduling policies, deciding where the job will reside and the processors local queue scheduling. The communication between the CPUM, the Launcher and the jobs is done through shared memory by control structures.

In order to get control of MPI jobs we use the Dynamic interposition mechanism. In particular we use the DiTools Library [7] that allows us to intercept functions like the MPI calls or a system call routine such that the Sginap, which is invoked by the MPI library when it is performing a blocking function. These functions provide information to the CPUM or gets information from them. In addition using this mechanism we can inhibit the execution of the Sginap routine. The Sginap wrapper, implemented as part of the CPUM library, is in charge of doing the context switching each time the spin time has expired and decides which process goes next. We use also this interposition mechanism to initialise some control structures of the CPUM runtime library and to find out each process MPI rank.

All the techniques were implemented without modifying the MPI library and without recompilation of the applications.

The CPUM wakes up periodically and at each quantum expiration examines if new jobs have arrived to the system or have finished execution and updates the control structures and

[2] We call MPL to the number of processes attached to a processor. When applying the Folding technique, each job will have its own maximum MPL depending on the Folding times.

if necessary depending on the scheduling policy, redistributes processors, context switch processes by blocking and unblocking them.

4.3 Processor Allocation Policies Evaluated

In order to do evaluations we compare our proposal, the FJT, with the Folding [3] policy and two Moldability policies: PSA and ASP [4]. We have implemented all of them and evaluated under several workloads varying machine utilization, Long and Short jobs proportion in the workload and job data sizes.

Table 1. Comparing characteristics of Processor Allocation policies

	PSA	**ASP-MAX**	**FOLDING**	**FJT**
Limit # Processors	ncpus	MAX	ncpus	ncpus
Job classification	no	no	no	yes
Queued jobs	Equipartition	-	-	Equipartition
Work-Conservative	no	no	yes	Both Work and Non-Work
Folding	no	no	yes	yes
Initial Folded times	-	-	1	4 (*Long* jobs)
Maximun MPL	1	1	2	4 (*Long* jobs)
Space Sharing	yes	yes	yes	yes
Processor Scheduling	-	-	Pure Time-Sharing	Co-Scheduling

In the table 1 above, we compare the main characteristics of each allocation policy implemented used for the evaluation. In ASP-MAX, the policy states a maximum number of processors to allocate while in the rest is just limited by the number of processors of the system (ncpus). About queued jobs ASP-MAX and FOLDING do nothing while PSA do an equipartition between all the queued jobs and FJT between Long Jobs if they exists otherwise between all queued jobs. PSA and ASP-MAX have always their maximum MPL equal to 1, while FOLDING and FJT may reach 2 and 4 respectively. Notices that while FOLDING initially start their jobs with MPL=1 that is unfolded, the FJT may start their Long Jobs with MPL=4, that is folded by 4.

5 Evaluations

Firstly we present the architectural characteristics of the platform used. Then we make a description of the applications and the workloads used.

5.1 Architecture

Our implementation was done on a shared memory multiprocessor, the SGI Origin 2000 [6]. It has 64 processors, organized in 32 nodes with two 250MHZ MIPS R10000 processors each. Each processor has a separate 32 Kb first-level instruction and data caches, and a unified 4 Mb second-level cache with 2-way associativity and a 128-byte block size. The machine has 16 Gb of main memory of nodes (512 Mb per node) with a page size of 16 Kbytes. Each pair of nodes is connected to a network router.

The operating system where we have worked is IRIX version is 6.5 and on its native MPI library with no modifications.

5.2 Applications and Workloads

To drive the performance evaluations we consider applications from the MPI NAS Benchmarks suite [5] and the Sweep3D. We have measured which portion from its execution time is dedicated to perform MPI functions in order to classify applications from high degree to low degree communication as shown in Table 2. The applications were run in isolation on 64 processors and having 64 processes each. For this article we have chosen just high degree

communication applications for being the most interesting and conflictive when sharing resources.

Table 1. % Time consumed for MPI Functions

Application	
cg.B.64	40 %
bt.A.64	46 %
sweep3D.64	47 %
lu.A.64	51 %

We have measured their linear execution time to determine the class they belong to: Long and Short jobs to help make decisions in our processor allocation proposal technique. In Table 3. we show this classification.

Table 3. Job classification depending on their linear execution time.

Long	Linear Execution Time
bt.A	2441 seconds
cg.B	4385 seconds

Short	Linear Execution Time
lu.W	177 seconds
Sweep3D	50 seconds

In [4] they show that Folding obtain better results when applications have a poor speedup. We will consider only the worst case as the NAS Benchmarks have demonstrated to scale well up to 64 processors. For this reason we will evaluate applications with a good speedup.

From the information above we arrived to the following workloads, all of them high communication degree, varying machine utilization and proportion of Long and Short jobs in the workload. The arrivals have an exponential distribution and the workloads were sized to 15 minutes of execution. These workloads were used to evaluate all the policies described in Table 1.

Table 4. Long and Short jobs proportion and machine utilization for each workload.

	% Machine utilization	% Long Jobs	% 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

6 Performance Results

Firstly we show the results for the evaluations of the policies in Table 1 under the workloads in Table 4. Finally we introduce arrivals by bursts and show the impact in the results obtained.

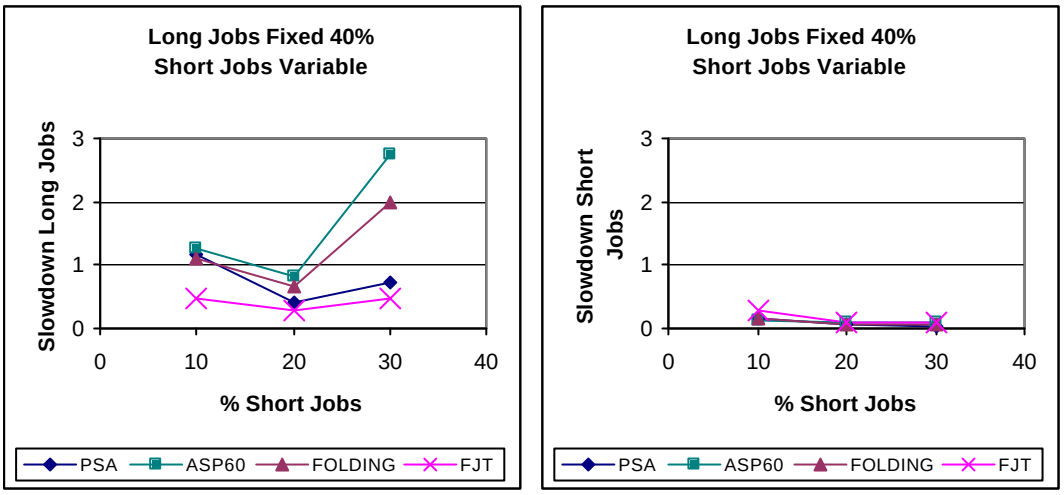

Fig. 1. Slowdown of the avg. resp. time with respect to FIFO execution for Long Jobs (left) and Short Jobs (right) respectively when varying % Short Jobs.

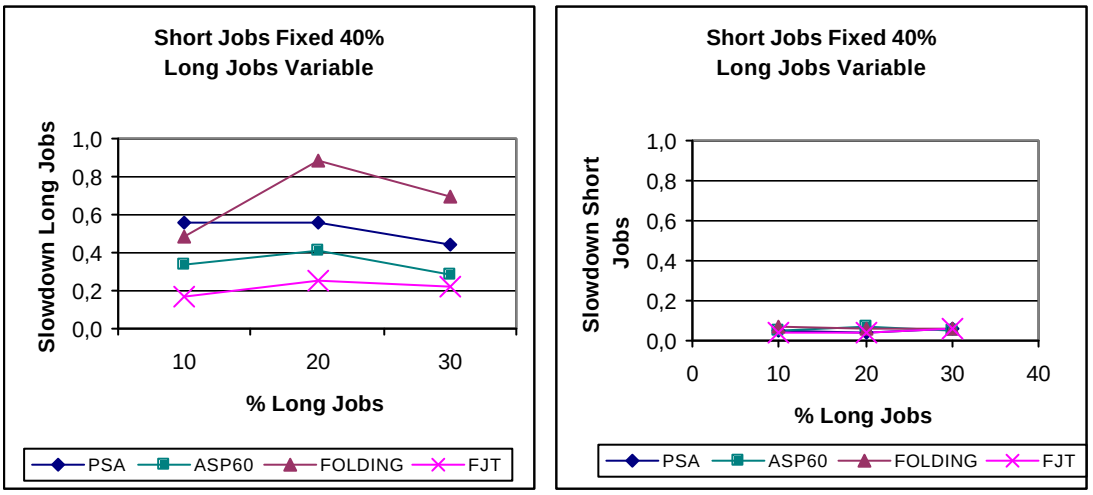

Fig. 2. Slowdown of the avg. resp. time with respect to FIFO execution for Longs Jobs (left) and Short Jobs (right) respectively when varying % Long Jobs..

In Fig. 1 we can see the slowdown of the average response time with respect to FIFO execution for Long and Short jobs when varying proportion of Short jobs in the workload. In Fig. 2 we can see the slowdown of the average response time with respect to FIFO execution for Long and Short jobs when varying proportion of Long jobs in the workload. Both proportions for machine utilization from 50 to 70%, distributed between Long and Short as described in Table 4. In the upper figures, Long Jobs are fixed in a proportion of 40 % of the workload and Short Jobs varies from 10 to 30%. In the bottom figures, Long Jobs varies from 10 to 30% and Short Jobs are fixed in a proportion of 40 % of the workload.

We can observe that when there are more Long than Short jobs, as in Fig.1, the performance for Long Jobs is better under FJT, especially for high machine utilization, while for Short Jobs it is very similar between policies.

Something similar happens when there are more Short than Long jobs, as in Fig.2 the performance for Long jobs is slightly better under FJT than under the rest of policies. Observe that FOLDING has the worst performance and PSA is quite similar for Long in Fig. 1. Another interesting observation is that Long jobs under JFT have almost the same slowdown no matter the percentage of Long or Short jobs.

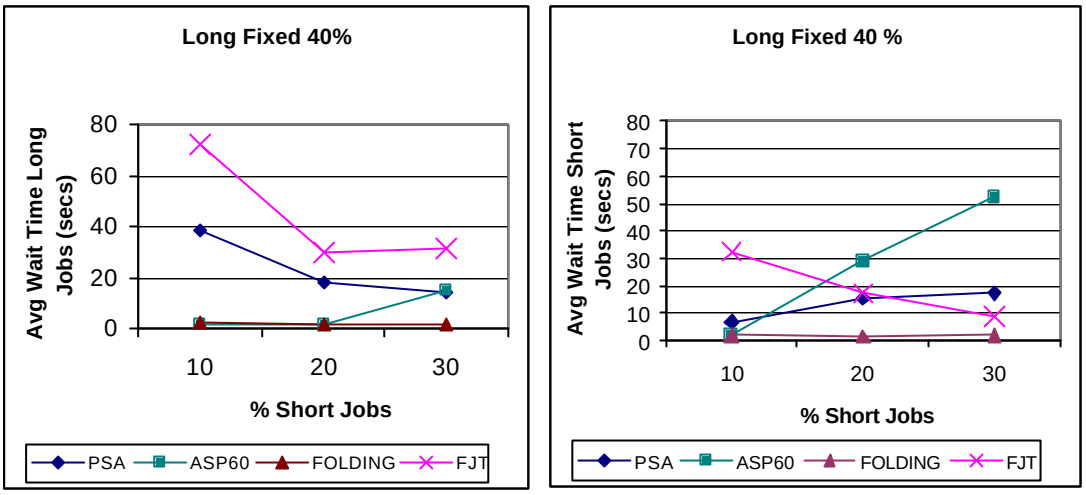

Fig.3. Average Wait-Time in the arriving queue for Longs Jobs (left) and Short Jobs (right) respectively when varying % Short Jobs.

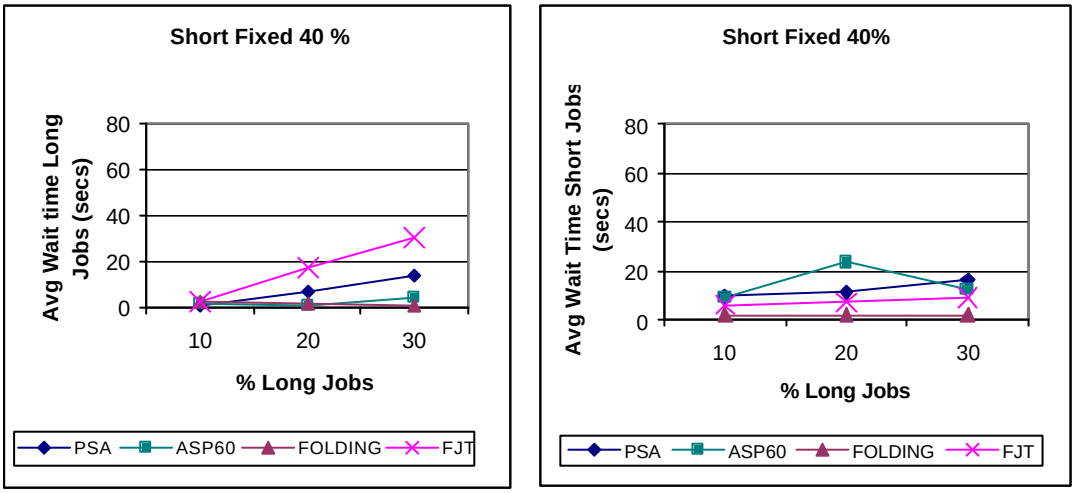

Fig.4. Average Wait-Time in the arriving queue for Longs Jobs (left) and Short Jobs (right) respectively when varying % Long Jobs.

In the figures above we can observe the average waiting times for Long and Short jobs when varying Short Jobs (upper figures) and Long Jobs (bottom figures). Something remarkable is that FOLDING achieves its objective of reducing the waiting time, it is the lowest. We obtain the greatest waiting times when having more Long Jobs than Short jobs (upper figures). This happens because when the arriving job queue is empty as in low machine utilization workloads, we delay the execution of jobs, in order to wait for more available processors if the waiting job is Long or to not bother the current execution if there are currently Long jobs folded running.

As shown in Figs 1 the PSA has similar performance to FJT. Remember that under PSA the system processors will be distributed among the queued jobs using equipartition. Let's study what happen when the workloads have arrival bursts. Under PSA, as bigger it is the burst, as fewer processors will be assigned to each job. After the burst, the system will be low loaded again, but the jobs won't be able to take advantage of this situation. Moreover if the burst have Short and Long jobs, the first ones will finish execution much earlier than the second ones freeing processors. So if Long jobs have initiated execution folded, at this time they could expand to the new available processors as under FJT.

In order to illustrate the situation we have run some synthetic experiments with a workload composed by a few Long and Short jobs, all starting at the same time as in a burst and running under FJT, FOLDING, PSA and ASP60 with burst of size 3, 6 and 9 applications.

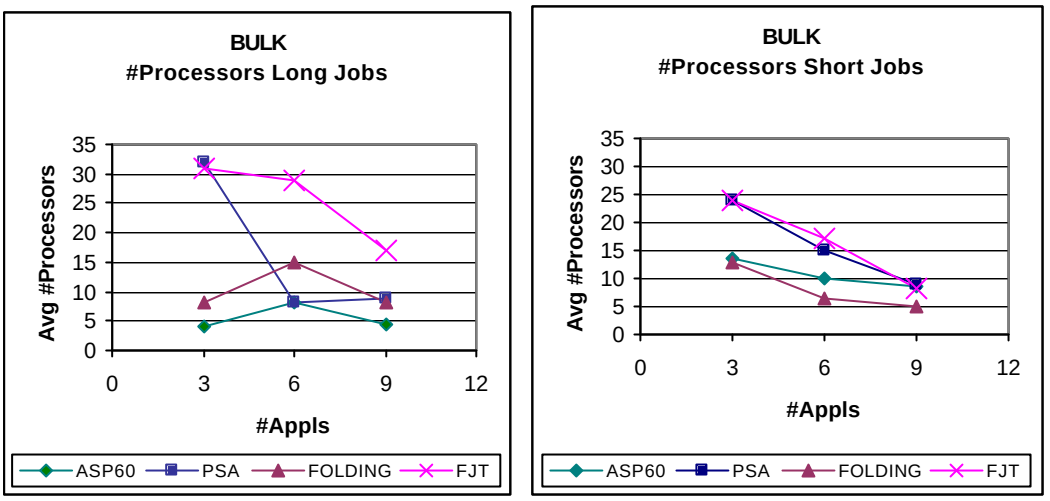

Fig. 5. Average number of processors assigned to Long Jobs (left figure) and Short Jobs (right figure) under each policy with a burst size of 3, 6 and 9 applications.

In fig. 5 we can see the average number of processors assigned to each application, under the different policies when the burst is of size 3, 6 and 9. Under Folding, PSA and ASP60 this number will be limited for the number of processors available at the start of execution. Under FJT as the job run folded at the beginning of execution its average number of used processors becomes greater when the burst has finished and the job has expanded. This is the case only for Long jobs, because the Short ones run always expanded.

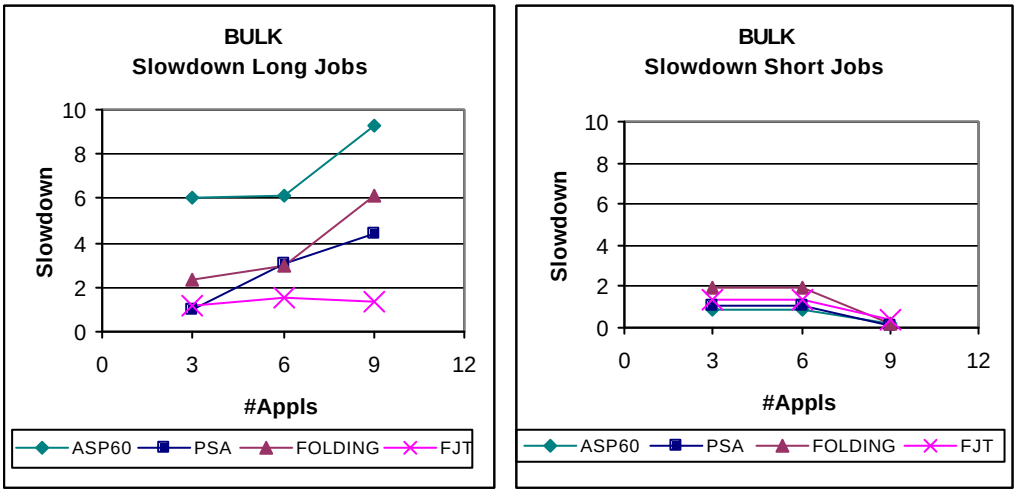

Fig. 6. Average slowdown of the response time of Long Jobs (left figure) and Short Jobs (right figure) under each policy with a burst size of 3, 6 and 9 applications.

In Fig. 6 we show the average slowdown of the response times for Long jobs (right figure) and Short jobs (right figure) with respect to FIFO execution. As expected, Long jobs have the smallest slowdown, especially when the burst has size 9. Moreover, it is possible to see that is quite the slowdown under FJT is quite constant, no matter the size of the burst. For Short Jobs, the slowdown is similar for all the policies. In spite of that the ASP60 has the best slowdown.

7 Conclusions

In this article we propose a processor allocation strategy based on Moldability and Folding techniques, the Folding by JobType (FJT). Our proposal differs from the existing ones in the decisions made for the choice of the initial number of processes, the number of Folding times, when to fold and at processor level, how processors will be shared, Co-Scheduling instead of pure Time-Sharing. In addition we treat jobs as belonging to two classes: Long and Short depending on the user submission execution time. So taking all this into account and

applying Folding only to Long Jobs, there are a lot of possible combinations, which we treated separately in order to design our proposal.

The proposed technique, FJT, is implemented and compared to an implementation of the Folding [2] which includes Moldability [3] and some pure Moldabilities techniques, like the ASP and PSA [4]. We evaluate them under workloads with different job sizes and classes, and machine utilization.

Results show that our proposed technique, the FJT, adapts easily to load changes. In general we obtain better performance than the rest of the techniques evaluated on high coefficient variation workloads. When there are more Long Jobs than Short Jobs, the JFT outperforms the rest of policies for Long Jobs. For Short jobs the performance between policies is quite similar. Our objective was to optimize execution time especially for Long jobs, not just waiting time as in Folding.

The proposal gets benefit especially when load goes from high to low peaks. As the jobs start execution folded, when there are available processors as the load goes down, the job will be able to expand to the new freed processors. This situation is very common in a workload with arrival bursts. We have study also this particular cases in some synthetic experiments. The results show that FJT has a great stability no matter the burst size. We obtained slowdowns of Long Jobs with respect to FIFO execution between 1.15 and 1.33 for burst sizes from 3 to 9. The PSA for example obtained for a burst size of 3 a slowdown of 1, but as the size of the burst incremented it reached a slowdown of 4.38 with respect to FIFO execution.

In the future we plan to extend the experiments to real workloads with arrival burst of different sizes.

We are planning to design a genetic algorithm that could learn from the recent past and present situation and taking into account the time the current jobs had been executing folded and unfolded, could decide the optimal number of processes and Folding times.

We believe also that performance for folded applications can be improved if the mapping would be done in a more intelligent way.

8 Acknowledgments

The research described in this work has been developed using the resources of the DAC at the UPC and the European Centre for Parallelism of Barcelona (CEPBA) and with the support of the CIRI in the “Donación para el CEPBA-IBM Instituto de Tecnología” project.

We would like to thank Xavier Martorell for his invaluable help and comments and, preliminary version of the CPUM.

9 Bibliography

[1] 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.

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

[3] 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.

[4] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, Lawrence W. Dowdy: *Analysis of Non-Work-Conserving Processor Partitioning Policies*. JSSPP 1995: 165-181

[5] 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.

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

[7] 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.

[8] G. Utrera, J. Corbalán and J. Labarta. "Study of MPI applications when sharing resources", Technical Report number UPC-DAC-2003-47 Dep d'Arquitectura de Computadors, UPC, 2003.

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