Scheduling of MPI Applications: Self Co-Scheduling

Gladys Utrera, Julita Corbalán, Jesús Labarta

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

Abstract. Scheduling parallel jobs has been an active investigation area. The scheduler has to deal with heterogeneous workloads and try to obtain throughputs and response times such that ensures good performance.
We propose a Dynamic Space-Sharing Scheduling technique, the Self Co-Scheduling, based on the combination of the best benefits from Static Space Sharing and Co-Scheduling. A job is allocated a processors partition where its number of processes can be greater than the number of processors. As MPI jobs aren't malleable, we make the job contend with itself for the use of processors applying Co-Scheduling. The goal of this paper is to evaluate and compare the impact of contending for resources among jobs and with the job itself.
We demonstrate that our Self Co-Scheduling technique has better performance and stability than other Time Sharing Scheduling techniques, especially when working with high communication degree workloads, heavy loaded machines and high multiprogramming level.

1 Introduction

An operating system must give support to different kind of parallel applications. The scheduler has to take into account the particular characteristics of each architecture and jobs to exploit the maximum performance of the overall system.

In our work, we will focus on shared memory multiprocessors (SMMs). This platform is widely used in computing centers with parallel machines. We will work on Message Passing Interface [6] (MPI) jobs. This message-passing library is worldwide used, even on SMMs, due to its performance portability on other platforms, compared to other programming models such as threads and OpenMP. This is especially important for future infrastructures such as information power grids, where resource availability, including platforms, dynamically changes for submitted jobs. Our objective will be to obtain the best performance in service time and throughput in multiprogrammed multiprocessors systems.

In this study, we try to demonstrate that it is possible to combine Co-Scheduling (CS) policies with Space Sharing policies to build a Dynamic Space-Sharing scheduling policy, which we call the Self Co-Scheduling policy (SCS), in a dynamic environment where the number of processors allocated to a job may be modified during job execution without loss of performance. We analyze several schemes that

exploit 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, as MPI jobs aren't malleable[1].

We show that jobs obtain better responses times and stability when share resources with themselves, as under SCS, than when share resources with other jobs, as under the Time Sharing techniques.

At processor level, we have observed that is better to free the processor as soon as the process detects it has no useful work to do and select the next process to run in the local queue, in a Round Robin manner.

The rest of the paper is organized as follows. In Section 2 is the related work. Then in Section 3 and 4 follows scheduling strategies evaluated and the execution framework. After that in Section 5 are the performance results. Finally in Section 6 the conclusions and future work.

2 Related Work

There are three main approaches that share processors among jobs: Time Sharing (TS), Space Sharing (SS), and Gang Scheduling or its relaxed version, CS.

When having more processes than processors, TS algorithms multiplex the use of processors among jobs in time. For parallel jobs the performance is degraded because of lack of synchronization and the context switching effect.

SS reduces the context switching effect, by partitioning the set of available processors among the jobs. These approaches have demonstrated to perform well on malleable jobs such as the OpenMP by adjusting the number of processes of a job to the number of available processors, which is not the case for MPI jobs.

As always the best choice comes from a combination: in this case from TS and SS: CS, classified in the literature as: Explicit Scheduling (ES), Local Scheduling (LS), and Dynamic or Implicit Co-Scheduling (DCS). In [8] there is an interesting classification of CS techniques based on the combinations of two components in the interaction between the scheduler and the communication mechanism: what to do when waiting for a message and what to do on message arrival.

The ES proposed by Feitelson and Jette [4] based on a static global list, execution order of the jobs and simultaneous context switching over processors, suffers from processor fragmentation, context switching overhead and poor scalability.

On the opposite, the scheduling decisions can be made independently at each local node, as in LS [5]. Here the performance of fine grain communication jobs is deteriorated because there isn't any synchronization at all.

DCS [1] is an intermediate approximation where decisions are made locally but based on global communication events such as message arriving and round trip message time. This information is combined in several ways, resulting in different schedulers. The Periodic Boost (PB) [8] is a kind of DCS where some monitor process manipulates the process priorities based on the unconsumed message queue. In most of these cases the MPI library was modified.

[1] Malleable jobs are the ones that can modify their number of processes at execution time.

With respect to coordination between the queuing system and the processor scheduler there is an interesting work in [3]. In [7] they determine at execution time the grain of the job. They classify an individual process as 'frustrated' if the scheduling policy actually applied doesn't satisfy its communication requirements.

3 Scheduling Policies Evaluated

We have implemented our SCS, the Static Space Sharing (SSS) and other existing TS techniques. SCS will be described in Section 3.1 and the TSs in Section 3.2.

Let's notice that under TS, there may be from all to none of the processes from a job executing at the same time. But under SCS there will be always at least a fixed minimum number of processes executing at the same time.

3.1 Self Co-Scheduling Technique

In order to combine SS and CS techniques we concentrate on three decisions: how many processors assign to a job, the process allocation in each partition and the scheduling of each processor local queue. These decisions are taken at low-level process management.

The number of processors assigned to a job is closely related to its number of processes and the multiprogramming level (MPL)2. This number is calculated by the equation (1):

$$\begin{aligned} &\textit{Initial\_allocation} = \text{Number Processes / MPL} \\ &\textit{Re\_allocation} = \textit{Initial\_allocation} + \\ &\qquad\qquad \text{(Number Free Processors / Number Running Appls)} \end{aligned} \quad \textbf{(1)}$$

Once the Queuing System launches a job, it starts execution if there are enough free processors that satisfy the *Initial_allocation* number given by **(1)**. 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 in an equipartition way, between the jobs in the system, using the Re_allocation number **(1)**. As soon as a new job arrives if there aren't enough free processors for it, all the jobs are shrinked to their Initial_allocation number of processors.

A user-level scheduler does the process mapping to the set of assigned processors in an ordered way using the internal MPI identification. As the number of processors assigned to a job could be less than its number of processes, there may be a process local queue at each processor, which we schedule by applying the CS techniques.

When executing a MPI blocking function, if the process cannot continue execution it may blocks immediately (BI) or do Spin Blocking (SB) [1]. After blocking a Context Switching routine is invoked. This decides which process of the local queue follows executing: the next in a Round Robin (RR) fashion or the process that has the greater number of unconsumed messages (Msg).

---

[2] We call MPL to the number of processes subscribed to a processor

So from these variations arise the following combinations of SCS: (1) With BI and RR; (2) With BI and Msg; (3) With SB and RR; (4) With SB and Msg. We consider Spin Times of 1200, 800 and 0 (this is BI) microseconds measured empirically.

3.2 Time Sharing Techniques

The TS techniques evaluated were: the scheduler of the IRIX 6.5 operating system as a pure TS scheduler and because is the native scheduler of the SMM where we are working on, an implementation of the Periodic Boost (PB) [8] for being the one that demonstrated the best performance in [8], and some variations to other CS techniques based on the actions taken on the message waiting and arrival.

The CS variations arise at processor level management. We have implemented the analogous combinations already described in Section 3.1 for SCS.

4 Evaluations

Firstly we present the architectural characteristics of the platform used. After that follows a brief description of the execution environment and the workloads.

The metrics used were the slowdowns of the response times with respect to FIFO execution, which means jobs executing in a first come first served order with a SSS scheduling policy.

4.1 SGI Origin 2000

Our implementation was done on a shared memory multiprocessor, the SGI Origin 2000 [9]. It has 64 processors, organized in 32 nodes with two 250 MHZ 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 asociativity 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 is IRIX 6.5 with its native MPI library with no modifications

4.2 Execution Environment

Our execution environment is composed by a queuing system: the Launcher and a user-level scheduler: the CPUM. 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.

Once the Launcher starts executing a queued job, it enters under the control of a user-level scheduler, the CPUM, which implements the scheduling policies. It decides how many processors allocate to a job, where it will reside , the processors local queue scheduling and controls the maximum MPL for all the applications in the

system, which is given as a parameter. The communication between the CPUM and the jobs is done through shared memory by control structures.

The CPUM wakes up periodically and at each quantum time expiration examines if new jobs have arrived to the system or have finished execution, updates the control structures and if necessary depending on the scheduling policy, redistributes processors. For a more detailed description about the CPUM mechanism refer to [10].

4.3 Applications and Workloads

To drive the performance evaluations we consider jobs from the MPI NAS Benchmarks suite [2] and the Sweep3D.

We have observed in initial experiments, that as we increment the MPL, the performance degrades due to the execution time dedicated to perform MPI blocking functions, like global synchronizations, barriers or blocking receives. We take this into account to construct the workloads and measure these percentages on a dedicated environment with 64 processes on 64 processors under the CPUM, with processes bounded to processors, to avoid external overheads.

For the first part of the evaluation, we have four workloads, from high *w1* (100% composed by high degree comm.. applications like CG, BT and Sweep3D) to low *w4* (50% composed by high and medium degree comm. applications like the MG and FT and 50 % composed by low comm. degree like the EP) comm. degree. And for the second part we consider only two types of comm. degree: high and low, but with arrivals time with exponential form distribution, sized to run for 5 minutes and with machine utilisation of about 60 % and 20 %.

5 Performance Results

Here we present the evaluations of the scheduling policies described in Section 3.

5.1 Evaluating Scheduling Policies under Different Workloads

The evaluations were run on 64 processors, MPL=4, with all the jobs having 64 processes each, and the same arrival time. In Fig. 3 we present the execution times of the workloads evaluated under the different policies.

As we can see in Fig. 3 the IRIX scheduler and the PB show the worst performance especially in high comm. degree workloads with slowdowns of 10 and 7 respectively.

A Spin Time equal to zero, that is BI seems the best option. This can be deduced from [1], as we have latency null because we are working on a SMM.

About the best slowdowns, we have SCS with 0.76 for high comm. and 0.58 for low comm. degree workloads. CS achieves 0.86 for high comm. and 0.72 for low comm. degree.

An interesting observation is that under SCS, the jobs execute always in the "same" environment, no matter the workload they are in, for being a kind of SS policy. But, under CS, applications become more sensible with respect to the

environment, because they must share processors between them. We measure the impact with the standard deviation coefficient (CV) of the execution times of the applications between workloads. For example the MG has under the SCS techniques a CV of 5.5, while under CS techniques it can be greater than 26.9. This means that SCS show a greater stability than CS. This is important because an application, under SCS, will have a more predictable behavior.

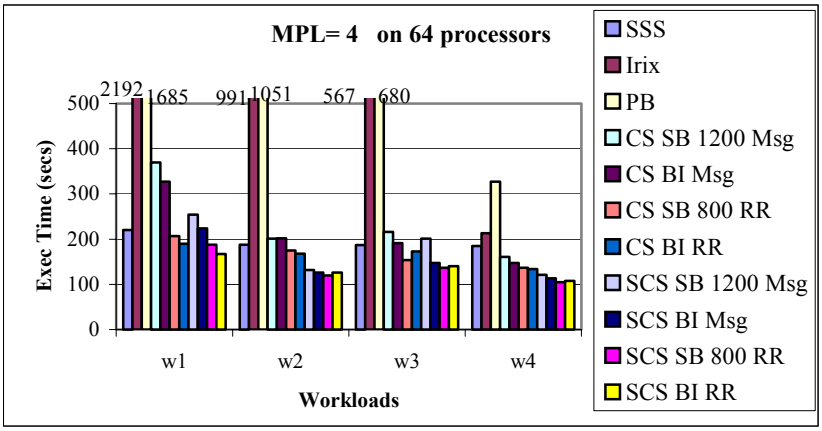

Fig. 1. Execution time in seconds for the workloads under different scheduling policies

5.2 Comparing CS and SCS varying workload and system characteristics.

In the last section we concluded that SCS with BI and RR dominated the other strategies evaluated. In this section we show the results from re-evaluating it and comparing with CS using the same configuration, with dynamic workloads, different machine load, MPL, communication degree, job sizes, number of processes (up to 60) and arrival time distributed in exponential form.

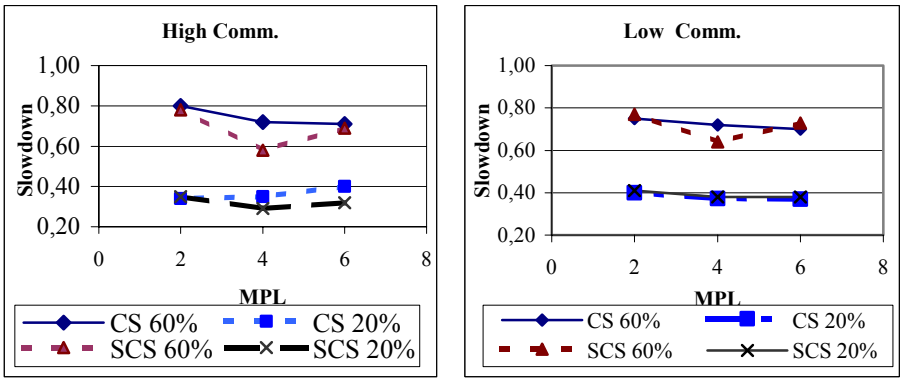

Fig. 4. Avg. Slowdowns of jobs within high and low comm.. degree workloads under SCS and CS with machine load of 20 and 60%

In [7] Moreira et al showed that MPLs greater than 6 are equal to infinite MPL. So the greatest MPL evaluated was 6.

In Fig. 4 we can see graphically the avg. slowdowns of the response times with respect to FIFO execution, for jobs in high and low communication degree workloads, evaluated for MPL=2, 4 and 6 under CS and SCS policies with machine loads of 20% and 60%.

In general SCS dominates CS especially for high comm. degree workload and MPL=4, in about 20%. For MPL=2, low comm. degree workloads and light loaded machines, SCS and CS are quite similar.

Our CS configuration generates some kind of unfairness, when high and low comm. jobs share resources, in favour of the second. This is because low comm. degree jobs, do very few context switches as they rarely invoke MPI blocking functions, so they almost never free the processor.

6 Concluding Remarks and Future Work

In this paper we have implemented and evaluated our proposed SCS technique and compared to SSS, and TS alternatives: PB, the native IRIX scheduler and some CS configurations, under different workloads and system characteristics.

In the first part of the evaluation to avoid fragmentation and strictly measure the impact with the maximum machine utilisation, we use a MPL=4, a fixed number of processes per job, equal to 64, and the same arrival time for the entire workload.

Here we concluded that the SCS techniques with BI and choosing the next to run in the local queue in a Round Robin manner seemed the best option. They showed also a more predictable behaviour and stability than the TS techniques. SCS have an average improvement from 20% to 40% over FIFO executions depending on the communication degree. On the other hand for applications, which have global synchronization or do mostly calculations, CS showed better response times.

In the second part, we re-evaluate and compare the SCS and CS techniques in a more realistic environment varying also machine utilisation, with different number of processes per job and arrivals time to the system.

We observed that as in the first part, SCS dominate the CS techniques especially for high machine utilisation. For low machine utilisation, CS and SCS perform quite similar.

There is some fragmentation under the SCS generated due to the MPL is fixed, which goes from 9% for MPL=6 to 24% for MPL=2. This can be avoided if MPI jobs were malleable. We are planning to work with moldable jobs, deciding the optimal number of processes to run the job, depending on its scalability and the current system fragmentation.

As MPL and machine utilisation increment, the fragmentation in SCS decrements, improving its performance. On the contrary, with high MPL, CS cannot afford the synchronization problem as good as the SCS.

In the future we plan to exploit the coordination between the queuing system and the processor scheduler, and the knowledge about the job obtained at execution time in order to determine for example its multiprogramming level dynamically.

Although this work has been done only on Shared Memory multiprocessors, we plan to extend this work to the other platform in the future.

7 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.

8 Bibliography

1. A.C. Arpaci-Dusseau, D. Culler. Implicit Co-Scheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. ACM Trans. Computer Systems 19(3), pp.283-331, Aug. 2001.
2. 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
3. J. Corbalan, X. Martorell, J. Labarta. Performance-Driven Processor Allocation. Proceedings of the 4th Operating System Design and Implementation (OSDI 2000), San Diego, CA, October 2000.
4. 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.
5. 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.
6. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. International Journal of SuperComputer Jobs, 8(3/4):165-414, 1994.
7. 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.
8. S. Nagar, A.Banerjee, A.Sivasubramaniam, and C.R. Das. A Closer Look at Co-Scheduling Approaches for a Network of Workstations. In Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99, Saint-Malo, France, June 1999
9. Silicon Graphics, Inc. IRIX Admin: Resource Administration, Document number 007-3700-005, http://techpubs.sgi.com, 2000.
10. 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.
http://www.ac.upc.es/recerca/reports/DAC/2003/index,en.html