Coscheduling under Memory Constraints in a NOW Environment*

Francesc Giné, Francesc Solsona
Departament d'Informàtica i Eng. Ind.
Universitat de Lleida, Spain
{sisco,francesc}@eup.udl.es

Porfidio Hernández, Emilio Luque
Departament d'Informàtica
Universitat Autònoma de Barcelona, Spain
{p.hernandez,e.luque}@cc.uab.es

Abstract

Networks of Workstations (NOW) have become important and cost-effective parallel platforms for scientific computations. In practice, a NOW system is heterogeneous and non-dedicated. These two unique factors make scheduling policies on multiprocessor/multicomputer systems unsuitable for NOWs. However, the coscheduling principle is still an important basis for parallel process scheduling in these environments.

We propose a new coscheduling algorithm for reducing the number of page faults across a non-dedicated cluster by increasing the execution priority of parallel tasks with lower page fault rate. Our method is based on knowledge of events obtained during execution, as communication activity and memory size of every task. The performance of our proposal has been analyzed and compared with other coscheduling implementations by means of simulation.

1 Introduction

The studies in [5] indicate that the workstations in a NOW are normally underloaded. This has invited researchers to develop different techniques in an attempt to adapt the traditional uniprocessor time-shared scheduler to the new situation of mixing local and parallel workloads [1, 10]. Basically, there are two methods of making use of these CPU idle cycles, task migration [6, 7] and time-slicing scheduling [8, 9]. In a NOW, in accordance with the research carried out by Arpaci [10], task migration overheads and the unpredictable behavior of local users may lower the effectiveness of this method.

In a time-slicing environment, two issues must be addressed: how to coordinate the simultaneous execution of the processes of a parallel job, and how to manage the interaction between parallel and local user jobs. One alternative is coscheduling [12, 14, 22].

*This work was supported by the CICYT under contract TIC98-0433

Coscheduling ensures that no process will wait for a non-scheduled process for synchronization/communication and will minimize the waiting time at the synchronization points. Thus, coscheduling may be applied to reduce messages waiting time and make good use of the idle CPU cycles when distributed applications are executed in a cluster or NOW system. Coscheduling decisions are made taking implicit runtime information of the jobs into account, basically execution CPU cycles and communication events [12, 13, 14, 15, 16, 22]. Our framework will be focused on an implicit coscheduling environment, such as scheduling the correspondents -the most recent communicated processes- in the overall system at the same time, taking into account both high message communication frequency and low penalty introduction into the delayed processes. The implicit property is also applied for coscheduling techniques that are not controlled by dedicated nodes or daemons.

However, the performance of a good coscheduling policy can decrease drastically if memory requirements are not kept in mind [2, 3, 11, 18, 19, 20, 21, 28, 29]. Most of them [2, 18, 21, 20] have proposed different techniques to minimize the impact of the real job memory requirements on the performance of a gang scheduling policy (original coscheduling principle, mostly applied and implemented in MPP's). However, to our knowledge, there is an absence of research into minimizing the impact of the memory constraints in an implicit coscheduling environment. We are interested in proposing implicit coscheduling techniques with memory considerations. That is to coschedule distributed applications taking into account dynamic allocation of memory resources due to the execution of local/distributed jobs by using implicit information (that obtained by observing local events in each cluster node).

In a non-dedicated system, the dynamic behavior of local applications (which consequently also varies its allocated resident memory) or a distributed job mapping policy without memory considerations cannot guarantee that parallel jobs have enough resident memory as would be desirable throughout their execution. In these conditions, the local

scheduler must coexist with the operating system's demand-paging virtual memory mechanism. In an uniprocessor system, paging improves memory and CPU utilization by allowing processes to run with only a subset of their code and data to be resident in main memory. However in distributed (cluster or NOW) environments, the traditional benefits that paging provides on uniprocessors may decrease depending on various factors, such as for example: the interaction between the CPU scheduling discipline, the synchronization patterns within the application programs, the page reference patterns of these applications [3, 28] and so on.

Our main aim is to reduce the number of page faults in a non-dedicated coscheduling system, giving more execution priority to the distributed tasks with lower fault page probability, letting them finish as soon as possible. Thus, on their completion, the released memory will be available for the remaining (local or distributed) applications. Consequently, major opportunities arise for advancing execution for all the remaining tasks.

However, the execution of the distributed tasks must not disturb local (or user) task interactivity, so excessive local-task response time should be avoided. It means that a possible starvation problem of this kind of tasks must be taken into account.

In this paper, a new coscheduling environment over a non-dedicated cluster system is proposed. The main aim of this new scheme is to minimize the impact of demand paged virtual memory, with prevention of local-task starvation capabilities. The good performance of this model is demonstrated by simulation.

The rest of the paper is organized as follows. In section 2, the main aim of our work is explained. Next, in section 3, the system model used is defined. A coscheduling algorithm based on this model is presented in section 4. The performance of the proposed coscheduling algorithm is evaluated and compared in section 5. Finally, the conclusions and future work are detailed.

2 Memory Constraints Motivation

Extensive work has been performed in the coscheduling area, but memory consideration effects in cluster computing performance have scarcely been studied. This fact, together with the different works done with coscheduling techniques [16, 23, 17], gives us a real motivation for an insight into coscheduling of distributed applications with memory constraints.

The execution performance of fine-grained distributed applications -those with high synchronization- could be improved by applying a generic coscheduling technique in cluster computing (section 2.1). However, the coscheduling techniques do not always improve performance. When task memory requirements in any particular cluster node overload the main memory, if virtual memory is supported by the o.s., the page fault mechanism (the swapper) is activated. The swapper interchanges blocks (the secondary memory transfer unit) of pages between the main and secondary memories. The swapper speed is usually at least one order of magnitude lower than the network latency. Thus, high or moderate page fault frequency in one node can drop distributed application performance drastically [3, 28, 29], overtaking widely in this way coscheduling benefits. In section 2.2, a solution for solving this situation is proposed

2.1 Coscheduling Benefits

How a coscheduling technique increases the performance of distributed applications and the page fault mechanism disturbs its progression is shown by means of the following real example.

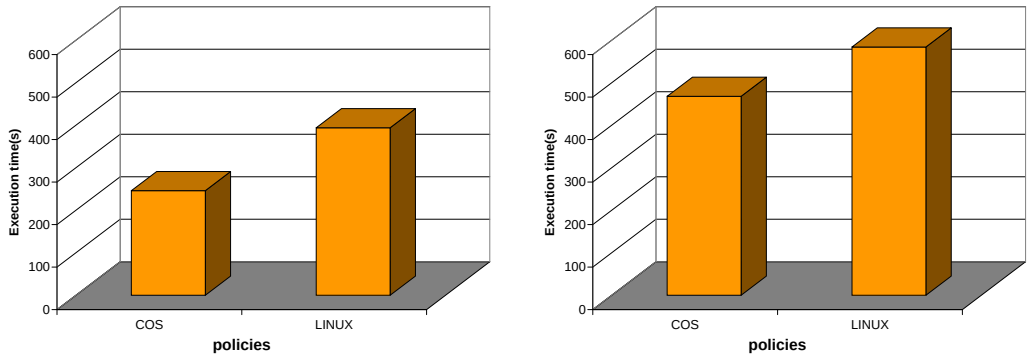
Two different environments were evaluated and compared between them, the plain Linux scheduler, (denoted as *LINUX*), and one implicit coscheduling policy, denoted as *COS* -for further information, see [17]- implemented in a real Linux cluster made up of 4 PC's with the same characteristics (350Mhz Pentium II processor, 128 MB of RAM, 512 KB of cache, Linux o.s. (kernel v. 2.2.14) and PVM 3.4.0). COS policy is based on giving more scheduling priority to tasks with more message sending and receiving frequency. The well known NAS [30] parallel benchmarks *MG* is used in this trial. Also, two synthetic local tasks have been executed jointly with the MG benchmark in each node.

Fig. 1(a) shows the good performance of the *COS* policy in relation to *LINUX* one in the execution of the MG benchmark when the main memory is not overloaded. As it was expected, COS policy gives priority the execution of the MG benchmark due to its intensive communication.

Fig. 1(b) shows the performance of the MG benchmark when it does not fit in its resident memory set because memory requirements of local tasks have been increased. As it can be seen on this figure, the page faulting mechanism (one or two orders of magnitude slower than the network latency) will corrupt the performance of distributed applications. This adverse effect is increased in applications with intensive communication because every page fault causes cascading delays on other nodes, thus decreasing the overall system performance. This fact points to the idea that memory requirements should be taken into account on the coscheduling processes with the aim of reducing the probability of page faults. This is certainly the main aim of this article.

2.2 Motivation

The following example (see Fig. 2) will help us to explain how the reduction of page fault rate is achieved and



(a) MG fits in the main memory

(b) MG does not fit in the main memory

Figure 1. Execution times (in seconds) for MG application

as, a consequence, the global performance is improved.

Let two intensive message-passing distributed applications, J_1 and J_2 and a cluster C made up of three homogeneous machines, N_1, N_2 and N_3 with a main memory size of *M* units. Each distributed application is composed of three tasks, each one is mapped in a different machine. Moreover, one local task is executed in every node. It is assumed that tasks memory requirements do not fit in the main memory of such node. The memory requirements of distributed tasks J_1, J_2 and local task (LOCAL) are denoted as m_1, m_2 and m_L, respectively. Figure 2 shows the contents of the swap memory (at the top) and main memory (in the middle) for node N_3 through the time. At the bottom, a two dimensional timing diagram representing the accumulative CPU time (in the Y axis) and the total executing time (executing plus waiting time, in the X axis) is also shown.

It can be seen that without any memory control policy, (Fig. 2 (a)) the local task is finished after 300 units, whereas task J_1 and J_2 finish after 450 units and 500 units, respectively. Figure 2 (b) shows the execution times obtained by applying a memory control policy consisting of giving more execution priority to the distributed task with the lowest page fault rate. That, in turn, means that the main memory space allocated to task J_1 -it has smaller memory requirements than J_2- is increased with time until it has all its address space residents in main memory, at the expense of memory space reduction for task J_2. Thus, task J_1 finishes its execution sooner than in case (a) (at time 350). When J_1 finishes execution, it frees its assigned memory and so the memory available for task J_2 is considerably increased, leading to a speedy execution of such task.

From the comparison of both techniques (fig. 2(a) and (b)), we can conclude that, by applying a memory policy control, the execution times for both distributed tasks, J_1 and J_2, have been reduced whereas local task one (task L) has been maintained. It is worth pointing out that although the progression of task J_2 during the first three periods is slower in case (b), when J_1 is finished, the J_2 CPU execution rate rises significantly because all the resources (CPU and memory) are available.

Note that this memory control policy should be applied with priority to distributed tasks with high synchronization requirements because, as synchronization grows more frequently, the impact of delaying any one node by a page fault is increased drastically. It suggests that the above memory policy should be applied in combination with a coscheduling technique based on communication activity. Section 4 explains how this coordination is achieved.

3 System Model

In this section, our model for cluster systems is explained. It provides for the execution of various distributed application at the same time. This model assumes that all the nodes in a non-dedicated cluster (or NOW) are under the control of our coscheduling scheme and also that the distributed applications are composed by a suite of tasks which are allocated and executed in the nodes making up the cluster. Also, every local coscheduler will take autonomous decisions based on local information provided explicitly or implicitly for the majority of existing time-sharing operating systems.

The model description is divided into two basic submodels, basic coscheduling and memory model. This way, the model will be more easily understood and a clearer separation between the basic underlying system requirements and the memory ones is performed.

In developing the model, various assumptions are made.

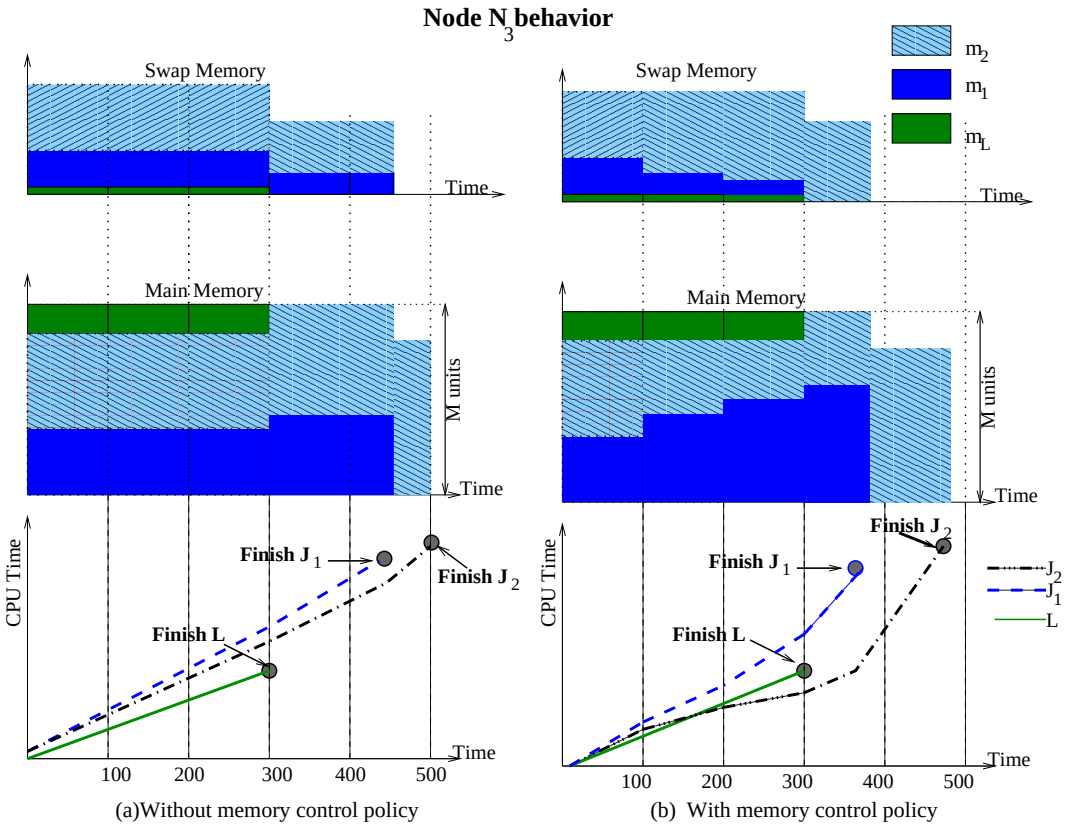


Figure 2. Coscheduling under memory constraints

Some of them are chosen with the aim of simplifying the model as much as possible. We are interested in developing a model that could be implemented further in a time-sharing o.s. Linux [25], due to its free property, fits in our purpose. Therefore, some assumptions are made taking Linux properties into account.

3.1 Basic Coscheduling Model

The model represents a cluster system for executing both distributed and local applications, so some preliminary assumptions must be performed. If different applications must be run in the cluster jointly, the underlying o.s. must be time-sharing. In a time-sharing cluster node, a task may be in different states (ready to run or simply ready, blocked, etc).

In this environment, tasks cannot communicate with their correspondents release the CPU before the expiration of their Time Slice and chance to the blocked state. It is assumed that incoming (out-going) messages to (from) a cluster node are buffered in a Receiving Message Queue, *RMQ* (Sending Message Queue, *SMQ*). This is a non-arbitrary assumption. For example, Unix-like systems [4], with the standard *de facto* TCP(UDP)/IP protocol, maintain (with particular differences) some sort of these buffers.

Let a cluster $C = \{N_k\}$, $k = 1...n$ and a task l, denoted as $task_l$, of node k (N_k). Every task belongs to a parallel/local job, denoted as *Job*. Next, some basic notation (and used in the remainder of this article) is summarized as follows:

- $RQ[k]$: pointer to the task on the position k of the Ready Queue (RQ). Special cases are k = 0 (*top*) and $k = \infty$ (*bottom*) of RQ. "*Top*" task is the currently executing task in the CPU and "*bottom*" is the latest one to be executed.
- $task_l.c\_mes$: number of current receiving-sending messages for $task_l$. It is defined as follows:

$$task_l.c\_mes = task_l.rec + task_l.send \quad (1)$$

,where $task_l.rec$ is the number of receiving messages for $task_l$ in the *RMQ* queue and $task_l.send$ is the number of sending messages for $task_l$ in the SMQ queue.

- $task_l.mes$: Past receiving-sending message number for $task_l$. This is defined as follows:

$$task_l.mes = P * task_l.mes + (1-P) * task_l.c\_mes, \quad (2)$$

where P is the percentage assigned to the past messages ($task_l.mes$) and $(1 - P)$ is the percentage assigned to the current messages ($task_l.c\_mes$). This field will be used to distinguish between local tasks ($task_l.mes = 0$) and distributed tasks ($task_l.mes \neq 0$). Based on experimental results [17], a $P = 0.3$ has been chosen.

- $task_l.de$: number of times that task l has been overtaken in the RQ by another task due to a coscheduling cause, since the last time such a task reached the RQ. This field will be used to avoid the starvation of the local and distributed tasks.

Note that some of this information is not explicitly maintained by the operating system, but can be easily obtained. For example, in a Linux o.s., $task_l$ will be represented by the *task_struct* structure (the Linux PCB, Process Control Block). The de field is not provided by the Linux o.s. but it could be added to the *task_struct* structure.

3.2 Memory Model

The Memory Management Unit (MMU) of each node is based on pages with an operating system which provides demand-paging virtual memory.

If the referenced page by a task is not in its *resident set* (allocated pages in the main memory), a page fault will occur. This fault will suspend the task until the missing page is loaded in the resident set. Next, such a task will be reawakened and moved to the RQ. Meanwhile another task could be dispatched for execution into the CPU.

The page replacement algorithm is applied to all the pages of the memory regardless of which process "owns" them (global replacement policy). Thus, the task resident size may vary randomly. Every node, taking into account some existing operating system trends, uses *the Last Recently Used (LRU) replacement algorithm*, in which the chosen page for replacement is the one that has not been referenced for a longest time. One example is the Linux o.s. that runs *the clock algorithm* [25] (a *LRU* approximation algorithm).

Some studies [24, 3] have shown the relation between the task page faults rate with respect to its resident set size and memory pattern access (*locality*). Locality is normally related to procedures, functions or some other kind of program-code association. The *working set* of a task [24], denoted as $task.wrk$, is a technique for obtaining an approximation to the locality. A task working set at time *t* with parameter τ is defined as the set of pages touched by the task during the last τ time units (*t*-τ,*t*). Large τ values can overlap various localities. On the other hand, low values for τ, may produce poor locality approximations [3].

We propose to calculate the page fault probability for each task. This way, it will be possible to determine which tasks have their locality resident in main memory. Low (high) page fault probability for a task will mean that its respective resident set fits (does not fit) its associated locality very well.

The proposed algorithm with memory constraints (see next section) will use the following notation (all of them use the memory concepts explained above):

- $task_l.vir\_mem$: virtual memory size for task l.
- $task_l.res\_mem$: resident memory size for task *l*.
- $task_l.nrpg\_fault$: number of times page faults have been detected for task l.
- $task_l.pg\_fault$: page fault probability for task l. It is computed as follows:

$$\begin{cases} 0 & if(task_l.wrk < task_l.res\_mem) \\ \frac{task_l.res\_mem}{task_l.wrk} & if(task_l.wrk \geq task_l.res\_mem) \end{cases} \quad (3)$$

- $N_k.M$: main memory size of the node k. Given that a homogeneous cluster is assumed, this parameter will be denoted simply as M.
- $N_k.mem$: memory requirements into node k. It is computed as follows:

$$N_k.mem = \sum_l task_l.vir\_mem \quad (4)$$

It is important to note that all the above fields are generally provided by the Linux operating system. For example, $task_l.res\_mem$, $task_l.vir\_mem$ and $task_l.nrpg\_fault$ are maintained in the *task_struct* structure, whereas $N_k.M$ is a global system parameter. Although Linux does not provide the working set of every task directly, it can be easily obtained from kernel space.

4 CSM: Coscheduling Algorithm under Memory Constraints

In this section, a local coscheduling algorithm with memory constraints (CSM, Algorithm 1) is proposed and discussed. Next, how CSM achieves the global coordination through the cluster is explained.

4.1 CSM Algorithm

The CSM algorithm must decide which task is going to run next, according to three different goals:

1. In a NOW, parallel applications must coexist with the operating system's demand-paged virtual memory. Paging is typically considered [18, 21] to be too expensive due to its overhead and adverse effect on communication and synchronization. So, one of the aims will be to minimize the number of page faults throughout the cluster.

2. The coscheduling of the communication-synchronization processes. No processes will wait for a non-scheduled process (correspondents) for synchronization/communication and the waiting time at the synchronization points will be minimized.

3. The performance of the local jobs. CSM algorithm should avoid the starvation of the local processes, minimizing the overhead produced by the execution of parallel jobs.

Algorithm 1 is implemented inside a generic routine (called *insert_RQ*). This is the routine chosen to implement our coscheduling algorithm because all the ready-to-run tasks must pass it before being scheduled. The *INITIALIZATION* section is the place where the different initializations (these may be global variables) are done. Note that an original *insert_RQ* routine should only contain one line of the form *(*$RQ[\infty] := task_h$*)*, which should insert $task_h$ at the bottom of the RQ.

If the RQ is empty (line 3), $task_h$ is inserted on the top, otherwise the algorithm works to find the position of the RQ where the task should be inserted in accordance with the starvation condition, communication rates and the page fault probabilities.

CSM is applied mainly to distributed tasks (those with $task_h.mes \neq 0$). It has no effect on the local tasks (generally tasks without remote communication, $task_h.mes = 0$), which are inserted at the bottom of the RQ ($RQ[\infty] := task_h$ in line 4).

The only way that CSM algorithm can know which task is currently executing in another node is taking the reception of the messages into account. For this reason, if the task to be inserted on the RQ has any incoming message in the RMQ queue ($task_h.rec \neq 0$) and the main memory in such node is overloaded ($N_k.mem > N_k.M$), the inserted task is led to the *top* of the RQ ($RQ[0]$). Thus, CSM applies a *dynamic technique* [15] to ensure that fine-grained distributed applications are coscheduled. In this technique, the more scheduling priority is assigned to tasks the more the receiving frequency is.

For the rest of the cases, the *SCHED_COND* function together with the *STARV_COND one* will establish the task ordering in the RQ.

From line 15 to 21, the scheduling condition is established according to memory constraints. If the memory requirements of all the resident tasks in such node ($N_k.mem$) exceed the main memory size (M), the *SCHED_COND* will depend on the page fault probability of the distributed tasks ($RQ[i].pg\_fault > task_h.pg\_fault$, line 17). This mode will be denoted as CSM mode. So, CSM gives more priority to the tasks with less probability of making a page fault. Taking into account the page replacement algorithm defined in the previous section, the pages associated with tasks with less chance of being scheduled will get old early. So, every time a page fault happens, the older pages will be replaced by the missing page. It means that with time, more memory resources will be allocated to tasks distributed with a lower page fault rate. In terms of locality, the local scheduler will settle for the resident set of such distributed tasks as can fit its locality.

Algorithm 1 CSM algorithm

1 ***procedure insert_RQ*** ($task_h$ *:task*)
2 *INITIALIZATION*
3 if($RQ[0] = NULL$) $RQ[0] := task_h$;
4 else if ($task_h.mes = 0$) $RQ[\infty] := task_h$; //Local task
5 else if(($task_h.rec \neq 0$) and ($N_k.mem > M$)) $RQ[0] := task_h$; //Dynamic mode
6 else
7 $i := \infty$;
8 while(STARV_COND(*i*) and SCHED_COND(*i*,$task_h$))
9 $RQ[i].de$ ++;
10 $i--$;
11 endwhile;
12 $RQ[i] := task_h$;
13 endif;
14 **endprocedure**

15 ***function SCHED_COND****(i:int,* $task_h$ *:task)* return boolean
16 if($N_k.mem > M$)
17 return($RQ[i].pg\_fault > task_h.pg\_fault$); //CSM mode
18 else
19 return($RQ[i].mes < task_h.mes$); //Predictive mode
20 endif;
21 endfunction

22 ***function STARV_COND****(i:int)* return boolean
23 **cons** $MNOL, MNOD$; //MNO≡ Maximum Nr. Overtakes (L:Local, D:Distributed);
24 if($RQ[i].mes = 0$) return ($RQ[i].de < MNOL$);
25 else return ($RQ[i].de < MNOD$);
26 **endfunction**

Otherwise, when memory requirements of all the tasks fit in main memory, another implicit coscheduling technique is applied. The condition shown in line 19 ($RQ[i].mes < task_h.mes$) is based on the *predictive technique*, [14, 23]. In predictive coscheduling, in contrast to the dynamic coscheduling, both send and receiving frequencies are taken

into account. The reason for doing it this way is that distributed tasks which only receive messages have normally less need to be coscheduled with their correspondents than those which perform both, sending and receiving.

The *STARV_COND* function, defined from line 22 to 26, avoids both the starvation of local tasks and distributed ones. For this reason, the inserting $task_h$ overtakes only the tasks whose *delay* field ($RQ[i].de$) is not higher than a constant named *MNOL* (Maximum Number of Overtakes) for local tasks or *MNOD* for distributed ones. The task field de is used to count the number of overtakes (by distributed tasks) since the last time the task reached the RQ. In line 9, this field is increased. The default value for *MNOL* is 2, as higher values may decrease the response time (or interactive performance) of local tasks excessively. Thus, starvation of local tasks (normally of the interactive kind and not communicating tasks) is avoided. It's worthwhile to point the necessity of evaluating also the field *de* for distributed tasks in order to avoid the starvation of a parallel job with a high page fault rate. For instance, if it wasn't taken into account, a continuous stream of parallel jobs with small memory requirements could provoke that a parallel job with a high page fault rate was always pushed to the end of the RQ. The value of *MNOD* constant will be evaluated experimentally.

How CSM algorithm maintains the coordination between the local decision takes in every node through the cluster is explained in the next section.

4.2 Global Coordination

As it has been explained above, CSM takes decisions locally. Also, depending on the characteristics of its resident tasks, it can work in different modes (Dynamic, Predictive or CSM). Thus, global coordination (through all the cluster) of the tasks making up distributed applications is performed in an independent manner in each cluster node, and depends on the three different execution modes:

1. Predictive mode: it establishes the task progression with high communication rates in nodes where main memory is not exhausted.

2. Dynamic mode: when the main memory is exhausted, CSM gives priority to the reception of messages in order to achieve the coscheduling between distributed tasks with high communication rates.

3. CSM mode: this mode works out in nodes where the main memory is exhausted. It schedules first tasks with lower page fault rate. This behavior favors tasks with locality fitting in its resident set. So jobs with smaller memory requirements will get priority implicitly with regard to jobs with bigger ones. Thus, in this situation, CSM behaves as a Small Job First policy

The above explained behavior could damage the performance of parallel jobs with little synchronization and high memory requirements. This is because CSM algorithm gives priority to the execution of jobs with small memory requirements in nodes whose main memory has been exhausted. CSM, by means of the starvation condition, avoids this situation. In the same way, distributed jobs with small memory requirements have associated a low execution time [2]. As a consequence, the memory released by these small jobs will be available for the remaining (local or distributed) applications sooner and then, they will be able to speed up its execution.

It's worthwhile to point out that decision taken by every local CSM scheduler depends on the ones made in the overall nodes of the cluster system. That is to say, a readjustment between the node modes in the overall system will be produced continuously.

There is an special case which requires particular attention. Suppose that CSM in one node gives priority to a task which belongs to a particular distributed application, while CSM in another node gives priority to a task which belongs to another distributed application. In this situation, the slowdown introduced in the implicated distributed application should be minimized by the starvation condition and the implemented dynamic mode, which maintains the synchronization between tasks.

Finally, note that the CSM behavior would lead to a FIFO coscheduling policy when all the distributed applications had similar memory requirements .

5 Experimentation

In this section, some simulation results are presented that compare the performance of the CSM policy, described in section 4, with other coscheduling algorithms. First, the simulation environment and the metrics used are presented. Then, the results obtained in the simulations are described and commented on.

5.1 Simulation and Metrics

Every node of the cluster has been simulated as shown in fig. 3. In this model, based on [26], when a tasks quantum is expired, the task is removed from the CPU and is reinserted in the RQ whenever the task has not finished all its requesting time. If a task does not expire its quantum due to a communication primitive or a page fault requesting service, it will be inserted in the SLEEP QUEUE. The local SCHEDULER will fix the order of the tasks in the RQ according to four different policies, a *round-robin policy* (RR), *a predictive coscheduling policy* (PRED), *a dynamic coscheduling policy* (DYN) and *a coscheduling policy with memory constraints* (CSM). The PRED policy will correspond to the

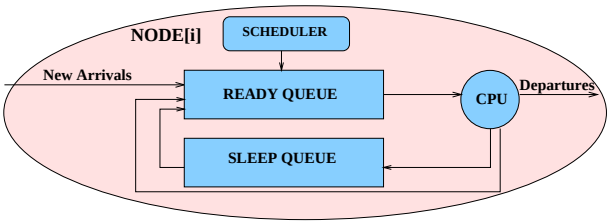


Figure 3. Simulation of a node

predictive mode of algorithm 1, whereas CSM policy will be the complete algorithm 1. DYN policy will correspond to the dynamic mode (line 5) described in section 4 without the memory constraints.

The chosen global simulation parameters are the following:

- Total processing requirement ($Job.tc$): the total processing requirement is chosen from a hyper-exponential distribution with mean $\overline{tc}$. It models the high variability that is expected in parallel supercomputing environments [27]. It is assumed that jobs can belong to three different classes - local tasks, small and large parallel jobs. *Small/local* and *large* jobs have a mean processing requirement of 300 and 3600 seconds, respectively. Each generated job is a *large* distributed one with probability *pdt*, and a *small* one with probability 1 - *pdt*. The density function is a Bernoulli with a *pdt* probability.

- Job size of distributed tasks ($Job.size$): job size is an integer that is calculated by 2^k within the range $[1, ..., n/8]$ for small jobs and $[n/4, ..., n]$ for large jobs, where n is the number of nodes in the cluster. It's assumed that a local task is a job with a $Job.size = 1$. By default, a value of n=32 is chosen. A static mapping of one distributed task per node is assumed. The mapping policy is based on obtaining an uniform memory load ($N_k.mem$) around the cluster.

- Mean inter-arrival time (*mit*): mean time for arriving distributed (local) tasks to the cluster (node). The chosen density function is a Poisson with mean = *mit*. The value for the *mit* parameter (for a predetermined average memory load of the cluster) has been calculated by means of the following equation:

$$mean\_load = \frac{mean\_size \times \overline{tc} \times (\sum_k N_k.mem)}{mit \times n^2 \times M} \quad (5)$$

where n denotes the number of nodes in the cluster, M the main memory size of a node and $mean\_size$ the mean job size, respectively. Note that a $mem\_load < 1$ means a mean memory requirements per node smaller than the main memory size (M), whereas a $mem\_load > 1$ denotes a mean memory requirements per node bigger than the main memory size (M). The $mean\_size$ is defined as follows:

$$mean\_size = \frac{\sum_k Job_k.size}{K} \quad (6)$$

where *K* is the number of executed jobs on the cluster.

- Memory size of the tasks ($task_l.vir\_mem$): an uniform distribution has been chosen for assigning a variable memory size (in page size units, the page size = 4KB) to each task in the range: [1,...,*mest*] for local, [*mest*,...,*2*mest*] for small and [*2*mest*,...,*4*mest*] for large tasks (*mes*t=8Kpages). Initially, the number of pages in the task resident set ($task_l.res\_mem$) will be computed according to the following equation:

$$\begin{cases} task_l.vir\_mem & if(N_k.mem < M) \\ \frac{(task_l.vir\_mem) \times M}{N_k.mem} & if(N_k.mem \geq M) \end{cases} \quad (7)$$

where $N_k.mem$ is defined according to the equation 4. However, during the simulation $task_l.res\_mem$ will be readjusted according to the scheduling frequency of a task and the page replacement algorithm explained in section 3.

- Mean service time (time slice): mean time in serving tasks (by the CPU). The chosen density function is an exponential with mean = 100ms.

- Message frequency ($task_l.mes$): the own message receiving-sending frequency is generated for each distributed task. The time between successive arrivals (sending) is simulated by means of an exponential distribution with mean=*mfreq*.

- Working set: in the 1970s, several empirical studies independently revealed important properties of program behavior [24]. These studies showed that programs tend to consist of *phases* which clearly dominate the fraction of total memory references and *transitions* which account for a considerable fraction of the program's page fault. The working set size during every *phase* is computed by means of a normal distribution with mean=$0.5 * task_l.vir\_mem$ for local tasks and a mean=$0.8 * task_l.vir\_mem$ for distributed tasks, according to the experimental results shown in [3]. The length of every phase will be calculated by means of an exponential distribution with mean=$0,1 * (task_l.tc)$. Thus a new working set will be established after every phase.

- Page fault probability ($task_l.pg\_fault$): every time that a task is scheduled in the CPU, a new page reference belonging to the working set of such task is generated. The density function is uniformly discrete. In

the case that the referenced page was not in its associated memory resident set, a new page fault would be generated. The page fault probability will be computed according to equation 3.

- Page fault latency: although the page fault latency can vary considerably depending on the cause of a page fault, in order to simplify it, a constant latency of 20 ms is assumed.
- Physical memory size (*M*): by default, a value of 32K pages is chosen.

The performance of CSM policy with respect to another coscheduling technique (RR and PRED) will be validated by means of three different metrics:

- Mean page fault number: it is defined as follows:

$$mean\_pg\_fault = \frac{\sum_k N_k.(\frac{\sum_l task_l.nrpg\_fault}{load})}{n} \quad (8)$$

  where n is the number of nodes of the cluster, $task_l.nrpg\_fault$ is the number of page faults and *load* is the number of executed tasks into node k.

- Correlation: this parameter shows how good is the coordination between decision taking by CSM algorithm in different nodes. It is defined as follows:

$$Correlation = 100 - \frac{\sum_k (\frac{Job_k.(tr_{fast} - tr_{slow})}{Job_k.tc})}{K'} \times 100 \quad (9)$$

  where K' is the number of parallel jobs executed into the cluster, $Job_k.tr_{fast}$ and $Job_k.tr_{slow}$ are the faster and slower response time of a task belonging to the job k, respectively. Note that according to the implemented mapping policy every node has at most one task belonging to a specific job.

- Slowdown: It is defined as follows:

$$Slowdown = \frac{\sum_k Job_k.tr}{\sum_k Job_k.tc} \quad (10)$$

  where $Job_k.tr$ and $Job_k.tc$ are the response time and execution time of the job k. For instance, let us suppose that 1000 jobs were executed in an experiment. The mean response time of these 1000 jobs was 50 minutes, and their mean execution time on processors was 25 minutes. Then, the Slowdown metric would be 2.

To sum up, the main goal of the CSM algorithm is to minimize the *mean_pg_fault* and *Slowdown* metrics and maximize the *Correlation* metric.

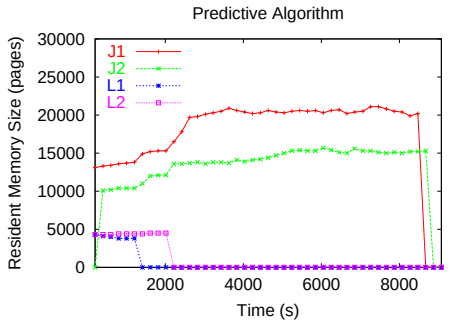
5.2 Evaluation of the CSM Performance

With the aim of verifying the good behavior of our simulator, a static workload made up of two *large* parallel jobs -J_1 and J_2- and two *local* tasks -L_1 and L_2- in every node of our cluster has been simulated. Traces of the evolution of the memory resident size ($task_l.res\_mem$) -at the top of fig. 4- and the progression of accumulative CPU time along the return time (CPU plus waiting time) -at the bottom of fig. 4- has been obtained with PRED and CSM algorithms for every task of one specific node (N_1). In this experiment, the communication frequency of both parallel jobs was initialized with the same value, attention being focused on the influence of memory constraints over the performance of the distributed tasks.

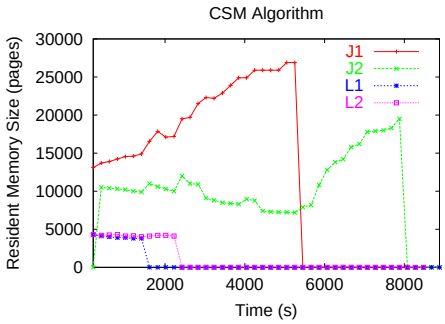
By analyzing the evolution of the CPU time obtained with PRED algorithm (fig. 4(c)), we see how distributed applications are considerably favored with respect to the local ones, as was expected. It must be taken into account that predictive algorithm gives more execution priority to tasks with higher communication frequency. The analysis of its memory resident size (fig. 4(a)) reflects how distributed tasks increase its memory resources and CPU accumulative time as soon as the execution of local tasks is finished.

Figure 4(b) and (d) reflects the behavior of CSM algorithm on node N_1. At first sight, it can be seen that the execution priority and the memory resources allocated to task J_1 are much bigger than those in task J_2. In fact, when task J_1 starts its execution, the overall memory requirements of such node are below 100%, and therefore all the initial memory resources requirements for J_1 are satisfied. When J_2 begins its execution, 10s after that J_1, the memory requirements in such node exceed 100% of the main memory and J_2 cannot fit its working set in its resident set. Therefore, as CSM gives more execution priority to tasks with lower page fault rate, J_1 advances execution faster than J_2. When J_1 execution finishes (after 5400s) -3072s before PRED case -, the J_2 accumulated CPU time rises sharply and proportionally to its memory resident size. For this reason, J_2 under CSM execution control has also a response time slightly better than with PRED case.

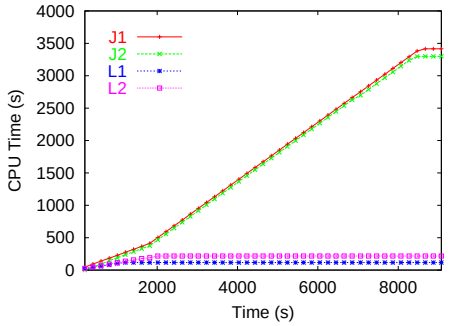
A dynamic workload has been used in next trials to verify the good performance of CSM algorithm. With this aim, three different environments has been simulated with a mem_load parameter equal to 0.7, 1.0 and 1.5, respectively. Every result shown in this section represents the average of 10 experiments, each of which runs 1000 jobs. This way, how CSM works under different memory constraints is analyzed in detail.



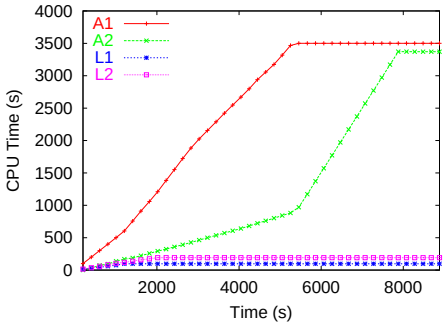
(a) res-mem for PRED algorithm



(b) res-mem for CSM algorithm



(c) CPU time for PRED algorithm



(d) CPU time for CSM algorithm

Figure 4. PRED vs CSM behavior on node N_1

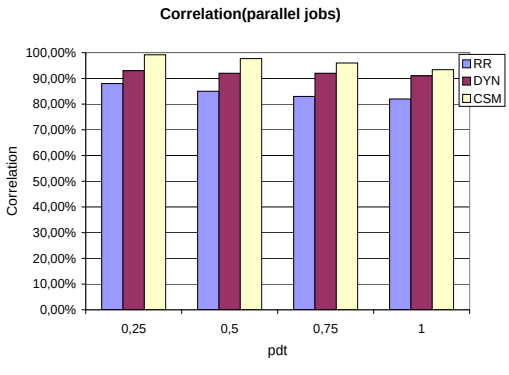


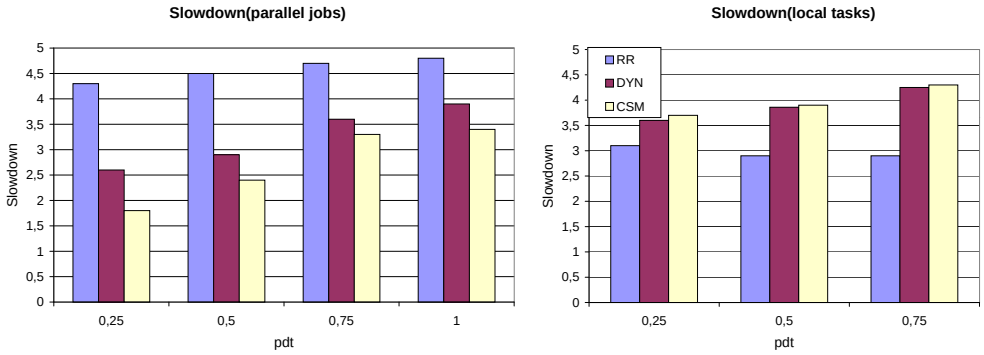
Figure 5. Correlation metric: CSM vs DYN and RR Algorithm

Analysis with low memory requirements

Small distributed jobs with pdt probability and *local* tasks with $(1-pdt)$ probability have been generated in this experimentation. Thus, a mem_load parameter equal to 0.7 corresponds with an average *load* per node equal to 2.8. In this subsection, the behavior of CSM algorithm is compared to the RR and dynamic (DYN) coscheduling algorithm. Note that under low memory requirements CSM algorithm works under the predictive mode explained in section 4.

Fig. 5 shows the level of correlation obtained with the three coscheduling policies. The three policies reach a high correlation (over 75%). While DYN is independent of the *pdt* probability and so of the kind of task (parallel or local), RR and CSM algorithms decrease with respect to pdt probability. RR correlation decreases with the rise of pdt because more parallel tasks are blocked waiting for a communication event, whereas CSM correlation decreases because in every node it can gives priority to a task belonging to different parallel jobs. For instance, while in node N_k job J_i could get top priority because it has the higher receive-sending message frequency, in the node N_{k+i}, another job J_j could get top priority over J_i for the same reason. Note that this situation is more plausible when more parallel jobs are executing concurrently in the cluster -pdt near 1-. However, CSM obtains the best correlation for all pdt values.

Fig. 6 (a) shows the obtained *Slowdown* metric for par-



(a) Slowdown for parallel jobs

(b) Slowdown for local tasks

Figure 6. Slowdown metric: CSM vs DYN and RR Algorithm

allel jobs with respect the pdt probability. The good performance obtained for the CSM mode demonstrates its effectiveness in making use of the coscheduling potential. This is due to the fact that the CSM technique takes both the current received-sending messages and the past messages into account whereas dynamic only works with the current received messages. This implies that more coscheduling chances are given in the CSM model. The high slowdown introduced by RR mode is a consequence of the low correlation reached by this technique.

Fig. 6 (b) shows the slowdown metric for local tasks. As it was expected, DYN and CSM techniques introduce a little more overhead than RR technique. It's worthwhile to point that CSM and DYN policies obtain similar results with this metric. The reason is that the number of delayed tasks is similar, and thus the introduced overhead is also equal. The dynamic policy increases executing priority of the distributed tasks fewer times than the CSM. However, when a distributed task increases its priority under DYN policy, it overtakes many more tasks than the CSM does (always moved on top of the RQ), and the consequence is that both methods introduce the same overhead.

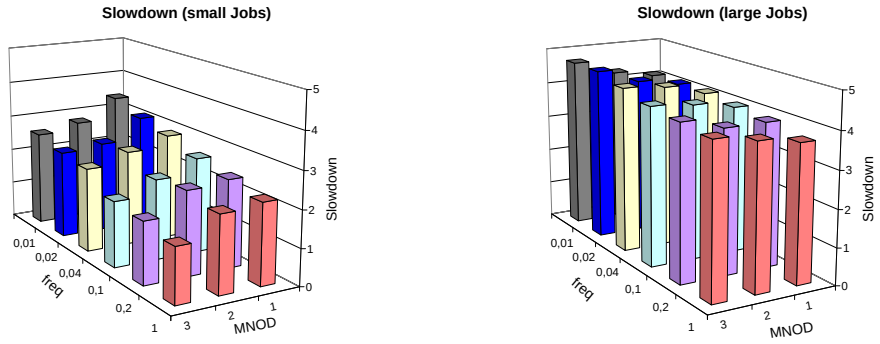
In [23] a detailed comparison of the predictive technique, used by CSM algorithm, with another coscheduling techniques by means of simulation can be found, whereas [17] shows a real implementation of the predictive coscheduling over Linux o.s. together with a detailed experimental analysis of its good performance.

Analysis with medium memory requirements

In this experiment a $mem\_load = 1$ has been chosen. This trial reflects the situation where memory requirements in some node can overload its main memory size due, for instance, to the activity of the local owner of such node. Thus, the three metrics (mean_pg_fault, correlation and slowdown) will be evaluated with respect to the message frequency for every implemented coscheduling policy: RR, PRED and CSM. In this trial, the three kinds of tasks (local, small and large) have been generated with a pdt value of 0.5. Taking this parameters into account, the simulated environment corresponds with an average *load* per node equal to 3.2.

Firstly, the influence of the MNOD constant (Maximum number of Overtakes for distributed tasks) over the performance of distributed tasks has been evaluated. Fig. 7 shows the slowdown obtained for small parallel jobs (fig. 7(a)) and large ones (fig. 7(b)) under different values of communication frequency and MNOD constant. Note how a MNOD constant greater than the mean load per node does not have any sense because a task can not overcome more tasks than mean required in such node. As it was expected, the slowdown for large jobs decreases with respect to the value of the MNOD constant. On the other hand, the performance of small tasks increases with the value of MNOD due to the fact that tasks with lower page fault rate have more opportunities for overcoming such distributed tasks with higher page fault rate. In both graphs, the slowdown decreases with respect communication frequency because CSM gives priority to the RQ top of such tasks with any waiting receive-send message and thus CSM favors intensive communication task execution. Taking this results into account, a $MNOD = 2$ has been chosen in this experimentation.

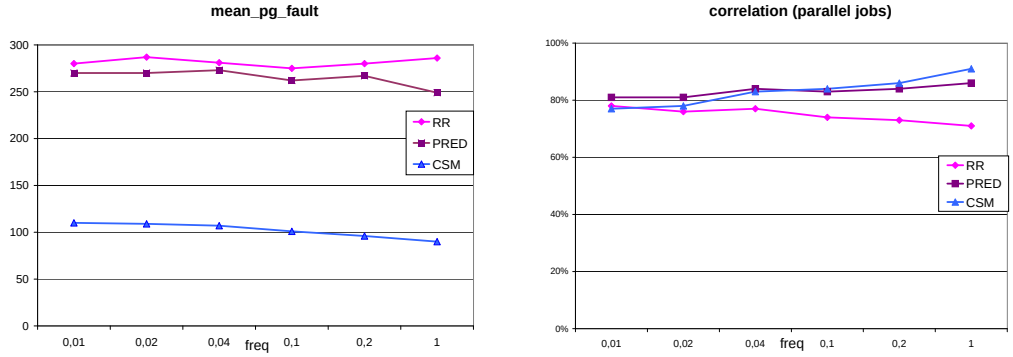
Fig. 8(a) and (b) shows the mean_pg_fault and correlation metric, respectively. The behavior of the mean_pg_fault reflects as CSM reaches totally its purpose of diminishing the number of page faults. The comparison of this parameter between RR and PRED algorithms



(a) slowdown for small jobs

(b) slowdown for large jobs

Figure 7. Variation of the slowdown metric with respect communication frequency and MNOD constant (mean_load=1)



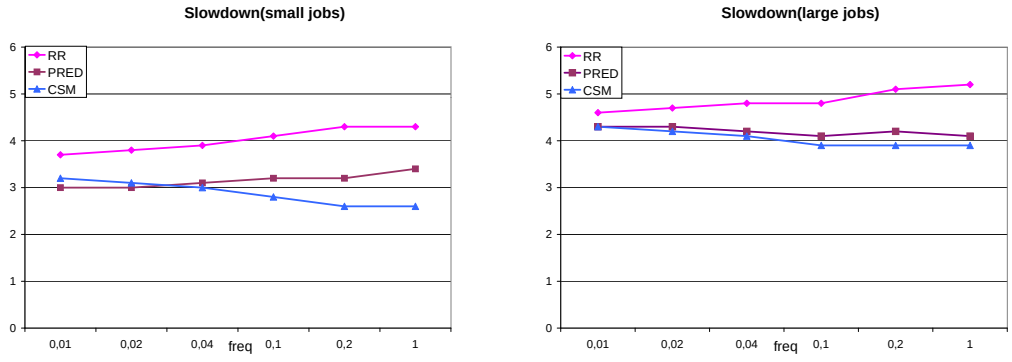
(a) mean-pg-fault

(b) correlation

Figure 8. mean_pg_fault and correlation metric (mean_load=1)

reflects that both makes a similar number of pages faults. The analysis of the correlation metric (fig. 8(b)) shows that CSM reaches the best result for intensive communication jobs whereas PRED correlation overcomes CSM correlation for non-intensive communication tasks. This is because CSM coordination under medium/high memory constraints is based on the current received messages whereas PRED takes an average receive-sending messages into account. As it was expected, RR reaches the worst coordination between nodes.

Fig. 9(a) and (b) shows the slowdown metric for small and large distributed jobs, respectively. For small jobs and as it was reflected in the correlation metric, CSM obtains the best results for higher communication frequencies whereas for lower communication frequencies, CSM and PRED obtain similar results. On the other hand, the performance of large jobs for the three coscheduling techniques is significantly worse. Large jobs performance feels more the effects of page faults and for this reason the slowdown is greater than for small ones. CSM and PRED techniques show a similar performance for large jobs. Although CSM, as it is reflected in fig. 8 (a), has a lower number of page faults than in other modes and so, it should be reflected in the gain of this algorithm, the mechanism of priority of this algorithm damages the performance of large jobs. Thus, CSM should take into account an agreement between the *MOND* value and the priority for lower page faults rate tasks.



(a) slowdown for small jobs

(b) slowdown for large jobs

Figure 9. Slowdown for parallel jobs (mean_load=1)

Analysis with high memory requirements

In this experiment, a $mem\_load = 1.5$, a $pdt = 0.5$ and a $MNOD = 3$ constant have been chosen. This trial reflects the situation, where memory requirements in the majority of nodes overload its main memory size. This simulated environment corresponds with an average *load* per node equal to 4.1.

Fig. 10 (a) and (b) shows the *mean_pg_fault* and *correlation* metric, respectively. The *mean_pg_fault* metric evolution reaffirms the good behavior of CSM algorithm pointed in the previous trial (mem_load=1) together with the poor results obtained by PRED and RR algorithms. The analysis of the *correlation* metric reveals how the negative impact of the page faults is increased for intensive communication tasks. As a consequence of the low page fault rate reached by CSM together with the coordination mechanisms implemented in such algorithm (see section 4.2), it obtains the best coordination between remote nodes for high memory requirements.

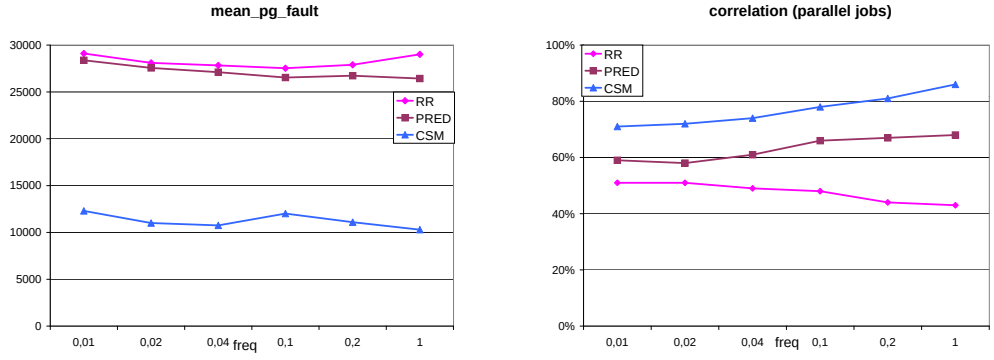
Figure 11 shows the slowdown metric for small jobs (a) and large parallel jobs (b), respectively. CSM reaches the best performance for small jobs as a consequence of its good obtained mean_pg_fault *and* correlation metric. This good trend is also reflected for large parallel jobs. Although CSM, as it was explained in section 4.2, favors the execution of the smallest jobs against the large ones, the interaction between low page faults rates and the *MNOD* constant implemented in CSM for avoiding the starvation of such kind of tasks, lead to the good behavior of CSM with respect to RR and PRED techniques.

Slowdown of local tasks under memory constraints

The local tasks slowdown obtained in both simulated environments above analyzed ($mem\_load = 1$ and 1.5) is shown in fig. 12 (a) and fig. 12 (b), respectively. In general, RR mode obtains the best results, although CSM and PRED results are very close to the RR ones. This is because of the influence of the implemented mechanism to avoid the starvation of local tasks. The delay introduced in the $mean\_load = 1$ case is not very high -in all the cases is lower than 4.25- and so the response time (the most important parameter for measuring interactivity) of the local tasks will be acceptable. Note that this difference will depend on the value assigned to the *MNOL* constant and the characteristics of the distributed tasks, as for example the communication frequency and memory requirements. It is worthwhile to point out that both coscheduling techniques (PRED and CSM) introduce the same overhead for local tasks, as both use the same mechanism to avoid local task starvation. As it was expected, when the $mean\_load$ is increased (see fig. 12 (b)) the slowdown obtained is worse. One aspect that is not reflected in this figure because our simulator does not take it into account, is the influence of the context switches over the local task performance. In this case, we think that the local tasks performance with CSM algorithm would slightly improve with respect to the other modes because the reduction of page faults obtained with this technique would provoke lower context switches and, as a consequence, an improvement of the performance.

6 Conclusions and Future Work

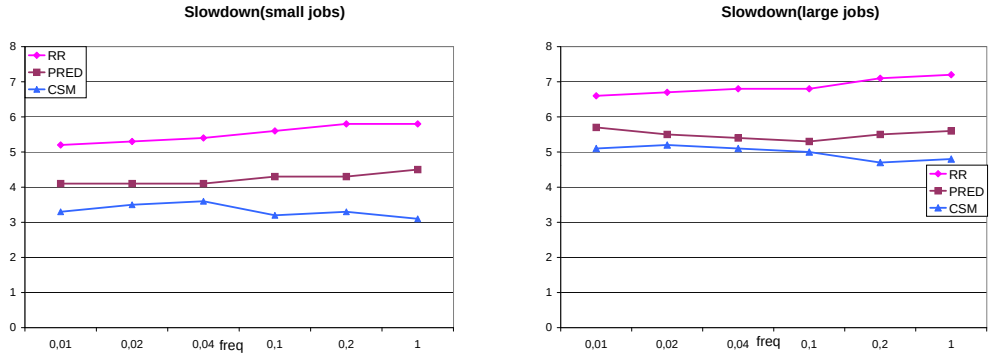
Demand-paged virtual memory attempts to optimize both CPU and physical memory use. The tradeoffs, which are well known for uniprocessors, are not nearly so clear for



(a) mean-pg-fault

(b) correlation

Figure 10. mean_pg_fault and correlation metric (mean_load=1.5)



(a) slowdown for small jobs

(b) slowdown for large jobs

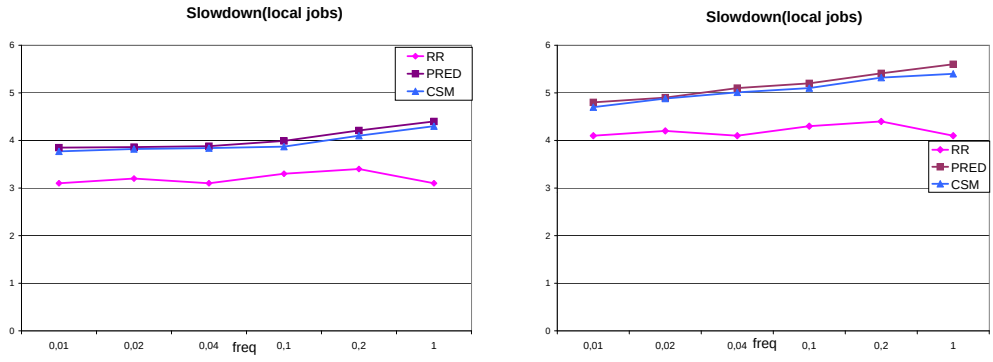
Figure 11. Slowdown for parallel jobs (mean_load=1.5)

NOW environments.

In this paper, a new coscheduling algorithm, abbreviated as CSM, for reducing the number of page faults across a non-dedicated cluster has been presented. CSM policy increases the execution priority of parallel tasks with lower page fault rates and simultaneously, it avoids local-task starvation. The performance of this proposal has been tested and compared with another coscheduling policies by means of simulation. The results obtained have demonstrated its good behavior, reducing the *Slowdown* of distributed tasks and maintaining the response time of local tasks with respect to another coscheduling policy.

According to the good performance of the CSM algorithm, future work will be directed towards investigating new coscheduling algorithms under memory constraints and implementing these in real PVM and/or MPI environments over time-sharing operating systems (such as the LINUX o.s.).

A new improvement to be introduced into our coscheduling system, would be to adjust the length of the quantum to the real necessity of the distributed tasks. This means that for parallel tasks with high CPU time requests, relatively coarse grain time sharing is probably necessary to provide good service to these jobs while not penalizing smaller jobs. So, our purpose would be to increase the length of quantum progressively by an amount proportional to context switching overhead. This way, our algorithm would amortize the context switch overhead associated to processes with large CPU requirements.



(a) slowdown for local tasks(mean-load=1)

(b) slowdown for local tasks(mean-load=1.5)

Figure 12. Slowdown for local tasks

References

[1] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik and P. Wong. "Theory and Practice in Parallel Job Scheduling". *In Job Scheduling Strategies for Parallel Processing*, D.G. Feitelson and L. Rudolph (eds), Lecture Notes in Computer Science, Vol. 1291, 1997.

[2] S. Setia, M.S. Squillante and V.K. Naik. "The Impact of Job Memory Requeriments on Gang-Scheduling Performance". *In Performance Evaluation Review*, March 1999.

[3] D. Burger, R. Hyder, B. Miller and D. Wood. "Paging Tradeoffs in Distributed Shared-Memory Multiprocessors". *Journal of Supercomputing*, vol. 10, pp.87-104, 1996.

[4] M. Bach. "The Design of the UNIX Operating System". *Prentice-Hall International Editions*, 1986.

[5] T. Anderson, D. Culler, D. Patterson and the Now team." A case for NOW (Networks of Workstations)". *IEEE Micro*, 1995.

[6] M. Litzkow, M. Livny and M. Mutka." Condor - A Hunter of Idle Workstations". 8th Int'l *Conference of Distributed Computing Systems*, 1988.

[7] S. Russ, J. Robinson, B. Flachs and B. Heckel. "The Hector Distributed Run-Time Environment". *IEEE trans. on Parallel and Distributed Systems*, Vol.9 (11). 1988.

[8] A.C. Dusseau, R.H. Arpaci and D.E. Culler. "Effective Distributed Scheduling of Parallel Workloads". *ACM SIGMETRICS'96*, 1996.

[9] M. Crovella et al. "Multiprogramming on Multiprocessors". *3rd IEEE Symposium on Parallel and Distributed Processing*, 1994.

[10] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson and D.A. Patterson. "The Interaction of Parallel and Sequential Workloads on a Network of Workstations". *ACM SIGMETRICS'95*, 1995.

[11] D.G. Feitelson. "Memory Usage in the LANL CM-5 Workload". In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 1291, pp. 78-84, 1997.

[12] J.K. Ousterhout. "Scheduling Techniques for Concurrent Systems." In *3rd. Intl. Conf. Distributed Computing Systems*, pp.22-30, 1982.

[13] F. Petrini and W. Feng. "Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems". *International Parallel & Distributed Processing Symposium*, Cancun, 2000.

[14] P.G. Sobalvarro and W.E. Weihl. "Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors". *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[15] P.G. Sobalvarro, S. Pakin, W.E. Weihl and A.A. Chien. "Dynamic Coscheduling on Workstation Clusters". *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[16] F. Solsona, F. Giné, P. Hernández and E. Luque. "Implementing Explicit and Implicit Coscheduling in a PVM Environment". *6th International Euro-Par Conference (Europar'2000)*, Lecture Notes in Computer Science, vol. 1900, 2000.

[17] F. Solsona, F. Giné, P. Hernández and E. Luque. "Predictive Coscheduling Implementation in a non-dedicated Linux Cluster". *To appear in 7th International Euro-Par Conference (Europar'2001)*, August 2001.

[18] S. Setia. "The Interaction between Memory Allocation and Adaptive Partitioning in Message Passing Multicomputers". In *IPPS Job Scheduling Workshop*, Apr. 1995.

[19] E. Parsons and K. Sevcik. "Coordinated Allocation of Memory and Processors in Multiprocessors". In Proc. ACM Sigmetrics/Performance'96, pp. 57-67, May 1996.

[20] W. Leinberger, G. Karypis and V. Kumar. "Gang Scheduling for Distributed Memory Systems". *6th International Euro-Par Conference (Europar'2000)*, Lecture Notes in Computer Science, vol. 1900, 2000.

[21] A. Batat and D. G. Feitelson. "Gang Scheduling with Memory Considerations". *Intl. Parallel and Distributed Processing Symposium*, pp. 109-114, May 2000.

[22] A.C. Arpaci-Dusseau, D.E. Culler and A.M. Mainwaring."Scheduling with Implicit Information in Distributed Systems". *ACM SIGMETRICS'98*, 1998.

[23] F. Solsona, F. Giné, P. Hernández and E. Luque. "CMC: A Coscheduling Model for non-Dedicated Cluster Computing". To appear in *IPDPS'2001*, April 2001.

[24] P.J. Denning. "Working Sets Past and Present". *IEEE Transactions on Software Engineering*, vol. SE-6, No 1, January 1980.

[25] M. Beck et al. "LINUX Kernel Internals". *Addison-Wesley*, 1996.

[26] L. Kleinrock. "Queuing Systems". *John Wiley and Sons*, 1976.

[27] D. Feitelson and B. Nitzberg. "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames IPSC/860". In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 215-227, April 1997.

[28] K.Y. Wang and D.C. Marinescu. "Correlation of the Paging Activity of Individual Node Programs in the SPMD Execution Model". In *28th Hawaii Intl. Conf. System Sciences*, vol. I, pp. 61-71, Jan 1995.

[29] V.G.J. Peris, M.S. Squillante and V.K. Naik. "Analysis of the Impact of Memory in Distributed Parallel Processing Systems". In *Proceedings of ACM SIGMETRICS Conference*, pp. 158-170, May 1993.

[30] Parkbench Committee. Parkbench 2.0. http://www.netlib.org/parkbench, 1996.