Coscheduling Techniques and Monitoring Tools for Non-Dedicated Cluster Computing*

Francesc Solsona[1], Francesc Giné[1], Porfidio Hernández[2] and Emilio Luque[2]

[1]Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain.
{francesc,sisco}@eup.udl.es

[2]Departamento de Informática, Universitat Autònoma de Barcelona, Spain.
{p.hernandez,e.luque}@cc.uab.es

Abstract

Our efforts are directed towards the understanding of the coscheduling mechanism in a NOW system when a parallel job is executed jointly with local workloads, balancing parallel performance against the local interactive response. Explicit and implicit coscheduling techniques in a PVM-Linux NOW (or cluster) have been implemented.

Furthermore, dynamic coscheduling remains an open question when parallel jobs are executed in a non-dedicated Cluster. A basis model for dynamic coscheduling in Cluster systems is presented in this paper. Also, one dynamic coscheduling algorithm for this model is proposed. The applicability of this algorithm has been proved and its performance analyzed by simulation.

Finally, a new tool (named Monito) for monitoring the different queues of messages in such an environments is presented. The main aim of implementing this facility is to provide a mean of capturing the bottlenecks and overheads of the communication system in a PVM-Linux cluster.

Keywords: coscheduling, monitoring tool, PVM, Linux, distributed and cluster systems.

1 Introduction

Combining parallel and sequential workloads on a non-dedicated Cluster system, with reasonable performance for both computation kinds is an open research goal. Researchers in this

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

area have shown that coscheduling techniques can offer this functionality, although the literature demonstrates that coscheduling is critical for parallel programs in order to achieving acceptable performance. This is an alternative to load balancing, which may be excessively time costly in Cluster computing. In addition, it may cause subsequent problems, such as redirection of messages (due to the migration of tasks), extra overhead and communication traffic (in managing and controlling the overall system), and so on [11].

Over the years, researchers have been developing time-shared distributed schedulers using coscheduling techniques, trying to adapt them to the new situation of mixing local and parallel workloads [1], [2], [3], [4] and [12].

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 [1]. Some of the relevant coscheduling work is shown below.

Explicit coscheduling, all processes in a parallel application are scheduled simultaneously, with coordinated time-slicing between them. Generally, this yields good parallel program performance and this is widely used to schedule parallel processes involving frequent communication [1]. Coscheduling will ensure that no process will wait for a non-scheduled process for synchronization/communication and will minimize the waiting time at the synchronization points.

Sobalvarro introduced in [5] and recently implemented in a Cluster system in [6], the concept of *demand-based* coscheduling, divided between *dynamic* coscheduling and *predictive* coscheduling. In *dynamic* coscheduling, messages arriving at a node, if addressed to a process other than the one currently running, sometimes cause preemption of the running process in favor of the process to which the message is addressed. The *predictive* method is based on coscheduling at the same time processes that have recently communicated with each other, but this is outwith the scope of this article.

A variation of dynamic coscheduling, named *implicit coscheduling* in [2, 3, 4, 7], is based on the spin-blocking technique. In this, the blocked process waiting for messages, spins for a determined time and if the response is received before the time expires, it continues executing. If not, the requesting process blocks and another one is scheduled.

Dynamic coscheduling, in contrast to implicit coscheduling, deals with all message arrivals (not just those directed to blocked processes), thus increasing the range of potential cases for coscheduling. In [6], only the execution of one distributed application was evaluated, due to the limitations of Fast Messages (a user messaging level under which a dynamic coscheduling algorithm was implemented), where in addition to providing dynamic coscheduling facilities for distributed applications, equal shares of the CPU for both distributed and local tasks were taken into account. In [7], implicit coscheduling was implemented (in a MPI [9, 10] environment) and evaluated, achieving performance for various coarse-grain message-passing distributed applications. More research needs to be done on dynamic coscheduling.

Algorithms for implementing new explicit and implicit coscheduling environments as well as studies of the parameters and overheads involved in them are presented in this paper. Also, a dynamic coscheduling model for cluster systems, named DCNDC, and some related dynamic performance metrics are explained. Multiple concurrent execution of distributed

applications is supported by this model. In this, it is assumed that Cluster nodes have to be mono-processor. A dynamic coscheduling algorithm based on this model is also proposed. Extensive performance analysis based on the model metrics presented, demonstrated by simulation, the applicability of both the proposed model and the dynamic algorithm in Cluster computing.

One of the most important goals in distributed computing and specially in PVM [8] environments is performance evaluation as Paradyn [18], Aims [15] and XPVM [14]. To study this, some questions must be answered: how good the message passing libraries of the distributed environment are, where there is room for improving their performance and so on.

The */proc* Linux file system offers much information about the communication subsystem, but this information is insufficient to obtain a global view of its behavior on each instant (bottlenecks, saturations, reasons for crashing in distributed applications, and so on). With this aim, a monitoring tool, called *Monito*, was designated to provide a means of investigating and localizing these phenomena.

The remainder of the paper is organized as follows. In Section 2, real explicit and implicit implementations are described. A dynamic coscheduling model and a dynamic algorithm based on this model are developed in Section 3. In Section 4, a monitoring tool for communicating system evaluation is presented. In Section 5, the different coscheduling techniques an tools are evaluated by real executions and by simulation. Finally, the conclusions and future work are detailed.

2 Explicit and Implicit Coscheduling

In this section, the methods and metrics to measure their cost for explicit and implicit coscheduling distributed tasks in a PVM-Linux NOW are described.

2.1 Explicit Coscheduling

The aim of explicit coscheduling is to schedule all the distributed tasks in the cluster at the same time and let them execute during a period of time. From one global controller process running in one node named *master*, control messages are sent (in a broadcast form) to every explicit process (named *dts*) running in the composing workstations of the cluster, which are responsible for implementing explicit coscheduling. One of these control messages (*init*) informs all the *dts* processes to start delivering STOP and CONTINUE signals to their local high-priority distributed processes at regular intervals (see also [12]). The time spent in starting (T_{start}) all the distributed tasks is:

$$T_{start} = W_s(local) + W_w(dts) + S_{sig}(CONT) + W_w(dis) + W_s(dts), \quad (1)$$

where W_w/W_s is the elapsed time in waking up/suspending *dts*, a local task (*local*) or a distributed task (*dis*). $S_{sig}(CONT)$ is the maximum elapsed time in sending a CONTINUE signal to all the distributed tasks in the node. The time spent in stopping (T_{stop}) all the

distributed tasks is:

$$T_{stop} = W_s(dis) + W_w(dts) + S_{sig}(STOP) + W_w(local) + W_s(dts), \quad (2)$$

where $S_{sig}(STOP)$ is the maximum elapsed time in sending a STOP signal to all the distributed tasks in the node. Because the time in delivering a signal to a group of processes does not depend on the signal to deliver, we consider that $S_{sig}(STOP) = S_{sig}(CONT) = S_{sig}$. Similarly, the values $W_w = W_s = W$ are considered to be equal. In consequence 1 and 2 can be reformulated as:

$$T_{ex} = T_{start} = T_{stop} = 4W + S_{sig}. \quad (3)$$

2.2 Implicit Coscheduling

The implicit coscheduling aim is to schedule only communicating distributed tasks at the same time. We are interested in only spinning the tasks during at most a context-switch period and not in spinning during the deliver of a round-trip message as in [2, 3, 4], as distributed tasks can follow many types of communication patterns and the messages can arrive asynchronously to distributed tasks, at any time. The metric T_{im} is used to compute the maximum overhead added in spinning, which also gives us a first reference to choose the *spin interval* (*sp*):

$$T_{im} = W_s(dis) + W_w(local) \quad (4)$$

3 Dynamic Coscheduling Model (DCNDC)

In this section, a Dynamic Coscheduling model for Non-Dedicated Cluster systems (named DCNDC) is formalized. Also a dynamic coscheduling algorithm (SDCA) for such a model is given.

Algorithm 1 shows the pseudo-code of the appropriative Round Robin Scheduling policy, for a time-shared o.s. of a cluster node, assumed in the DCNDC Local Scheduler. Other typical scheduler functions (like saving/restoring task contexts) that do not influence our model are not considered. The scheduler works indefinitely while the RQ (sorted Ready to run task Queue) wasn't empty. In this case, the scheduler dispatches (assigns the CPU to) the top task.

When the executing task finishes execution, or its assigned time slice expires, or another task preempt the CPU due to a dynamic decision, the execution in the CPU continues in line 4 (label DISPATCH), where some dynamic statistical information of such task must be carried out after this point.

Algorithm 2 shows the pseudo-code of the proposed dynamic coscheduling algorithm (SDCA). The INITIALIZATION and RESIDUAL CODE sections are the place where the different initializations (these may be global variables) of the algorithm and where the rest of the original source of the routine receive reside respectively. It is assumed that message

Algorithm 1 DCNDC Local Scheduler of a cluster node.

```
1 do forever
2   if (number of processes in RQ ≠ NULL)
3     dispatch (first process of RQ);
4   DISPATCH:
5     accounting for the CPU out-coming process;
6     move CPU out-coming process to the bottom of RQ;
7  endif;
8 enddo;
```

reception (*m*) is done asynchronously, so we called *async_receive* the entry point for such a routine. Note the action of reception routine only differs from the original one (with the *async_receive* function and the RESIDUAL CODE section) when the receiving tasks are not executing. This means that dynamic coscheduling techniques will only be applied when this condition was satisfied. In line 7, the receiving process, according to a dynamic condition is re-ordered (moved) into the RQ. The dynamic condition applied are the following:

- number of received messages (a process can overtake another one if has more pending messages in the Message Buffer Queue (*MBQ*) than such a last)
- maximum number of overtaking (a process can not overtaking another one if this maximum is reached by such a last)

Algorithm 2 Share Dynamic Coscheduling Algorithm (SDCA).

Routine Receive

```
1 INITIALIZATION
2 async_receive (message, process):
3    if (process ∉ RQ) insert_RQ_bottom (process); endif;
4    RESIDUAL CODE
5     mark process as potential coscheduling;
6    if (process ≠ first RQ process)
7       move process in the RQ according to a dynamic condition;
8       increment the overtaking field for the overtaken processes
9       if (process reaches top position in RQ) Goto DISPATCH; endif;
10   endif;
```

4 Monito: The Monitoring Tool

Monito (a *Moni*toring *to*ol), provides different facilities for collecting and analyzing the communication subsystem in a PVM-Linux cluster. Monito obtains on time state of the main queues involved in the communication process, from the PVM (version 3.4) to the physical network device layer.

Table 1: Netmon arguments.

$Netmon -d*sp* -t*mt* [-s -f] [Interface]
-d *sp* : sampling period (*sp*) in milliseconds
-t *mt* : monitoring time (*mt*) in seconds
-s : output to display
-f : output to file Netmon.dat
[Interface] : sampling interface, default eth0

The most interesting transmission/reception queues to be analyzed in each layer are *hosts, locltasks and txlist/pvmrxlist* in the PVM, *write_queue/receive_queue* in the socket, *buffs/backlog* and *CBL/RFA* in the logical and physical device respectively.

The set of implemented utilities are: two PVM services, *pvm_getpvmdstats* and *pvm_getaskstats*, the *stadsoc*, *stadque* and *staddev* modules, the *dev_queues* system call and finally *Netmon*, an application that monitors and collects information about these utilities.

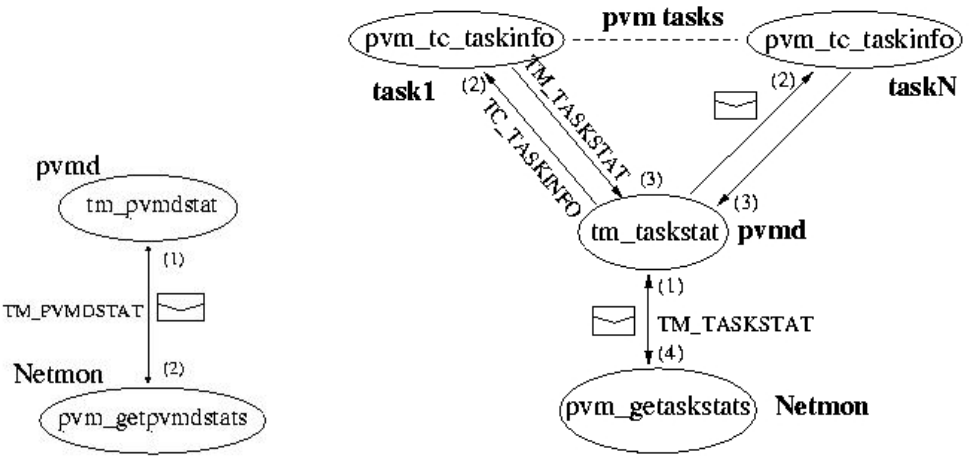
4.1 Netmon

The *Netmon* arguments are shown in Table 1. Netmon does the following operations in every *sampling period* (*sp*):

1. Obtain PVM information

   (a) Obtain *pvmd* (PVM daemon) statistics. It is carried out by the pvm call *pvm_getpvmdstats* (see Fig. 1(a)). The function *pvm_getpvmdstats* sends a TM_PVMDSTAT message to the PVM daemon and waits for a response from it (1). In the daemon a new function, *tm_pvmdstat* was implemented to reply *to Netmon* with another TM_PVMDSTAT message containing the information of the *hosts* and *locltasks* structures (2), like, for example, the packets to deliver to remote hosts (in *hosts*) and packets to deliver to the local tasks (in *locltasks*).

   (b) Obtain PVM tasks statistics. It is begun by the new pvm call *pvm_getaskstats* (see Fig. 1(b)). The function *pvm_getaskstats* sends a TM_TASKSTAT message to the daemon and waits for a response from it (1). In the daemon, a new function for dealing with this kind of messages was implemented, named *tm_taskstat*. This function sends a TC_TASKINFO message to all the pvm tasks (2). Next, this function waits for the reply from all the pvm tasks by a new *pvm_tc_taskinfo* function (3) and then sends a TM_TASKSTAT message to *Netmon* (4). The information obtained is the number and size of the buffered messages, waiting for sending in *txlist* (or to be taken in *pvmrxlis*t) queues.



(a) (b)

Figure 1: (a) *pvmd* monitoring; (b) pvm tasks monitoring.

2. Obtain Linux information

   (a) Obtain the sockets statistics. Netmon reads the file */proc/net/stadsoc*, created and maintained by the *stadsoc* module for storing *write_queue* and *receive_queue* information. Table 2 column stadsoc, shows the information provided by this module. Note that this information is also supplied by the kernel in three different */proc* files but the overhead in reading these can be unacceptable in small sampling periods. This is the reason for implementing this facility.

   (b) Obtain logical device statistics. The method used to get information about the *backlog* and *buffs* queues of the logical devices is the same as in the previous explained module. This module is named *stadque* and its associated file is */proc/net/stadque* (see Table 2 column stadque). There is no known utility that gets this kind of information.

3. Obtaining network device information. To capture additional information of the physical network device (see Table 2 column staddev), not supported in the */proc/net* file system and also to sample their RFA and CBL queues, another module, *staddev* has been implemented. Its associated file is */proc/net/staddev*.

Note that the PVM data is collected by message passing; this can produce some monitoring overhead. When it finalizes, Netmon shows on the display the additional *Netmon* execution, the percentage of samples which overlapped the *sampling period* and the maximum extra time required in a *sampling period*.

Table 2: stadsoc, stadque and staddev information.

stadsoc	stadque	staddev
protocol type (tcp, udp, raw)	# of queues	received collision packets
@IP and port Source	max. queue length	pending packets in RFA
@IP and port Target	Interactive queue	delayed transmission packets
sk_buff's in recv_queue	Normal queue	# one trans. collision
sk_buff's in write_queue	Background queue	# multiple trans. collisions
total bytes in recv_queue	# of backlog sk_buff's	pending packets in CBL
total bytes in write_queue		
# of retransmissions		
i-node socket		

5 Experimentation

The experimentation has been performed in a Now made up of an interconnection network of 100 Mbps Fast Ethernet and four PVM-Linux PCs with the same characteristics: 350Mhz Pentium II processor, 128 MB of RAM and 512 KB of cache.

A distributed application, *sintree* was implemented to measure performance of the implemented environments. It attends for a communication pattern of one to vary, and vary to one. *sintree* accepts two arguments: number of processes (*M*) and number of iterations (*N*). By default $M = 4$ and $N = 30.000$. Also, two kernel benchmarks (class A) from the NAS parallel benchmarks suite [13] were used: *is* and *mg*.

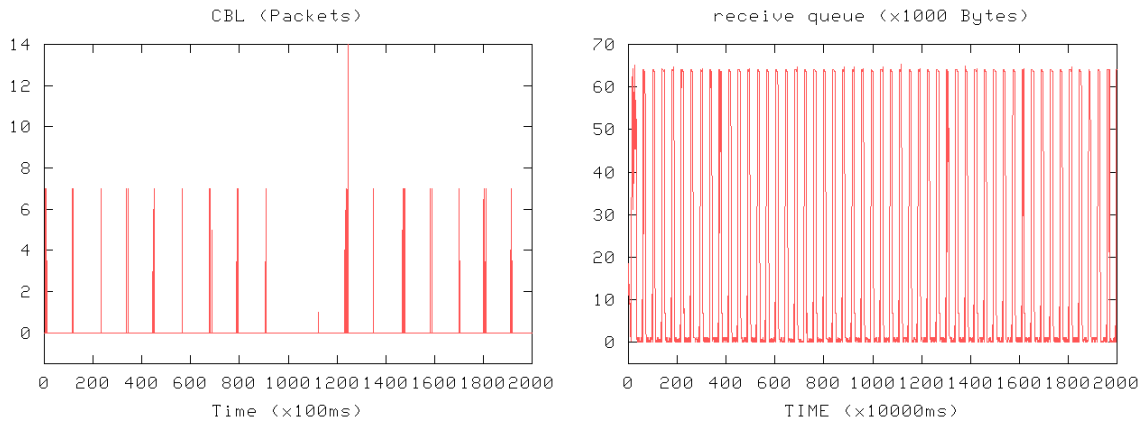
Firstly, the well behavior of the implemented monitoring tool is evaluated. Next, different explicit and implicit coscheduling implementations are compared by means of experimental results. Finally, the performance of the dynamic algorithm explained in the section 3 is evaluated by means of simulation.

5.1 Monito Experimentation

In every experimentation, the PVM operation mode *(RouteDirect* or *DontRoute* [8]) and arguments of the sintree benchmarks (#processes/size_of_messages) were showed. For example, 32p/8K means that *sintree* arguments are N = 32 processes and M = 8KB. The default *Netmon* arguments were $sp = 100\mu s$ and $mt = 200s$. In the figure 2 and 3 are showed the most representative results obtained from the parent node of the sintree benchmark .

5.1.1 Physical, Logical and Socket Layers

Fig. 2(a) shows the results obtained for the physical layer in the DontRoute 32/2MB case. The CBL queue is fulfilled due to the great number of fragmented packets transfered from higher levels. The maximum CBL and RFA capacity is 16 packets, but for security reasons, the driver always reserves two CBL elements. For this reason, the maximum number that appears in Fig. 2(a) is 14.



(a) (b)

Figure 2: DontRoute (a) buffered packets in transmission (CBL queue) for 32p/2MB and (b) *pvmd* socket buffer in reception (*receive_queue*) for 750p/8K.

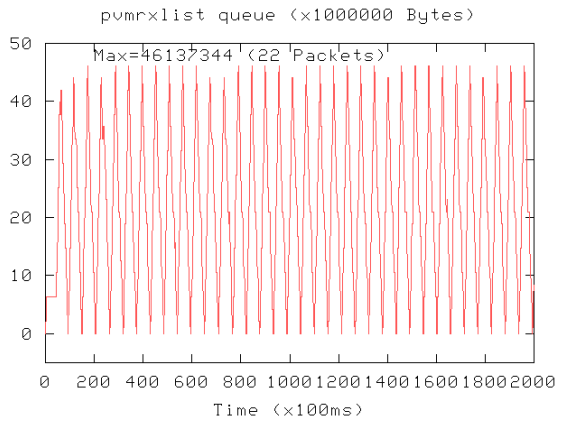


Figure 3: RouteDirect *pvmrxlist* queue 25p/2MB.

Fig. 2(b) shows the socket layer statistics for the DontRoute 750/8K case. Note that the reception queue is saturated (the maximum capacity is 65535 bytes). This confirms the good behavior of *Monito*.

There is no buffer saturation or relevant facts in the other cases of these layers and thus the results obtained are not shown.

5.1.2 PVM Layer

Fig. 3 shows the reception queue (*pvmrxlist*) in the parent task of the *sintree* benchmark (almost saturated). Observe the result of dividing the max. *pvmrxlist* capacity reached (= 46137344 Bytes) by the number of packets (= 22, number of crosses in one maximum of

the figure) is 2097152 Bytes = 2*MB* (exactly the sending message size); this also proves the good behavior of *Monito*.

5.2 Explicit and Implicit Experimentation

In order to compare different coscheduling algorithms, the next distributed environments were created:

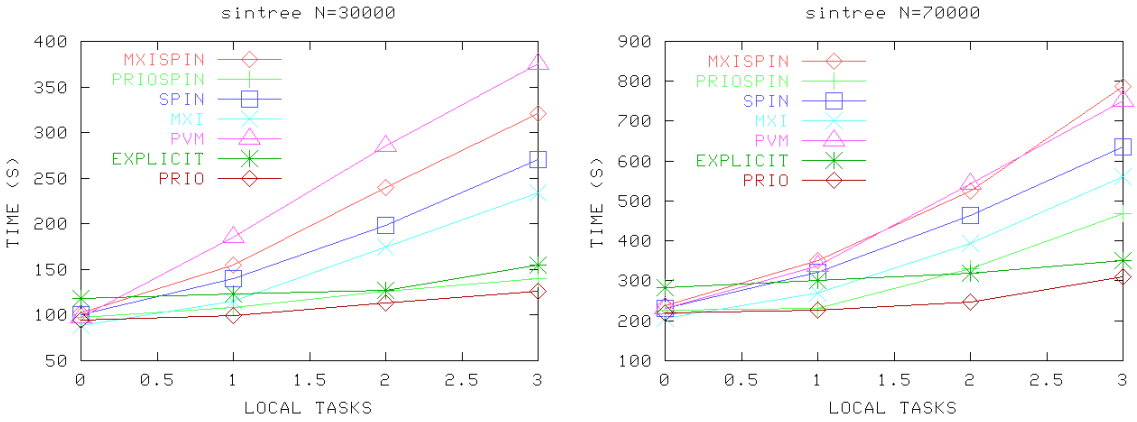
- PVM: original PVM.
- SPIN: Implicit coscheduling (the spin-block is only performed in the reading of the data fragment).
- MXI: head + body fragment are read in one time (original PVM does this in two different steps).
- MXISPIN: SPIN and MXI.
- PRIO: always, the highest priority is assigned to distributed tasks.
- PRIOSPIN: PRIO and SPIN.
- EXPLICIT: periodically, after 90000 μs the *dts* daemon in each node delivers a STOP signal to all the local distributed processes and then, elapsed 10000 μs, *dts* delivers a CONTINUE signal to reawaken them. The measured $T_{im} \simeq 10$ μs, so in the spin models an *sp* of 10 μs was chosen. In all the experimentations, the communications between remote tasks was done through *RouteDirect* PVM mode.

5.2.1 Distributed tasks performance

Fig. 4 shows the *sintree* execution times in the seven above cited modes while the local workload in each node (simulated by compiling applications) is varied from 0 to 3.

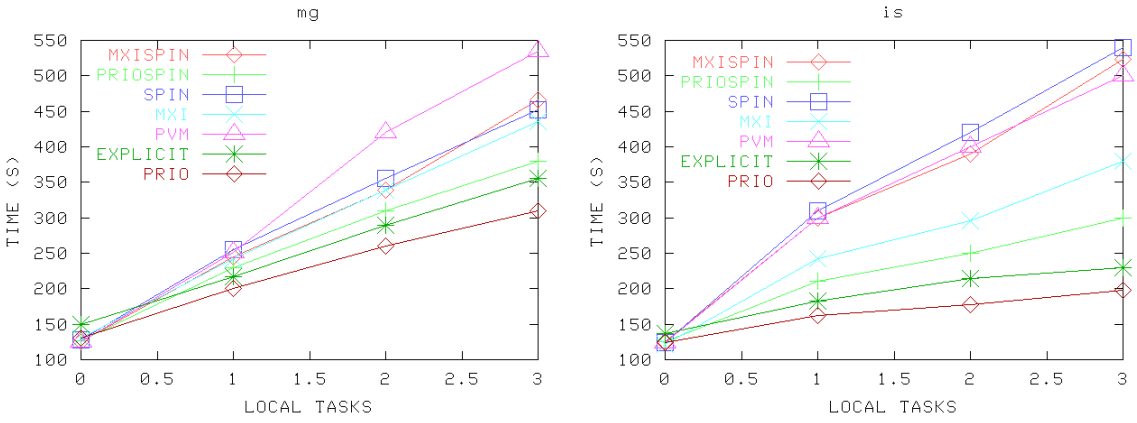
As was expected, optimal execution of the PRIO case can be observed. EXPLICIT without local tasks is the worst mode. By increasing the workload, its performance scarcely decreases due to the time assigned to distributed tasks is independent of the local workload. MXI and SPIN modes scale fine and their performance is always between the PVM and PRIO. SPIN is faster than PVM because avoids a lot of times the blocking overhead in receiving messages. The PRIOSPIN case gives worse results than PRIO, as the unnecessary spin-block phase added in the first mode, this only adds an unnecessary overhead in the reading of the fragment. MXISPIN works worse than MXI, as in this case penalties when time-slice expires are more than ones in context switching.

Fig. 5 shows the results obtained from executing *is* and *mg* in the different models. The behavior of *mg* is similar to the *sintree* one. On the other hand, *is* does not work as fine as *mg* and *sintree* in the SPIN cases.



(a) (b)

Figure 4: *sintree* execution. (a) N = 30000. (b) N = 70000.



(a) (b)

Figure 5: Execution of the NAS parallel benchmarks (a) *mg* and (b) *is*.

5.2.2 Local tasks performance

The influence of the models in the local tasks was based on measuring the slowdown calculated as follows:

$$sd_{MOD} = \frac{TMODEL - TPVM}{TPVM} 100,$$

where TMODEL (TPVM) is the execution time of a local task (a compiling application) when it was executed in such model (original PVM). See Table 3.

Table 3: slowdown of a compiling local task.

slowdown	PRIO	PRIOSPIN	EXPLICIT	SPIN	MXI	MXISPIN
sintree	1.4	1.4	1.4	3.6	1.4	3.6
is	2.8	2.8	4.2	2.1	0	2.1
mg	90	92	42	8	1.6	8

As might have been expected, when intensive message-passing distributed applications are executed (*sintree* and *is*), the local task performance is scarcely decreased. On the other hand, a high slowdown is introduced if intensive CPU distributed tasks are executed (*mg*). As was to be expected, the explicit model has a great impact on the local task and even more in the PRIO and PRIOSPIN cases.

5.3 DCNDC Experimentation

The experimentation has been performed by simulation. Three different simulating programs have been implemented. The first program simulates the SDCA algorithm (SDCA), explained in the section 3. The second one simulates the dynamic technique (DYNAMIC) defined in [5] and implemented in [6]: in the receipt of a message for a task *l*, if *executing_cpu_cycles(l) + share < executing_cpu_cycles(e)*, where *share* is a constant of the system and *e* is the task actually in execution, then a context switch in favor of task l is performed. Finally, the last one simulates the spin-block technique (IMPLICIT) explained in the section 2.2.

The following parameters have been considered for implementing these simulating programs:

- mean inter-arrival time (*mit*): mean time for arriving tasks to the RQ (Ready Queue), simulated by means of an exponential distribution with mean = *mit*.
- mean service time (*mst*): mean time for service tasks (by the CPU). Also, the chosen density function is an exponential with mean = *mst*. A value of *mst* = 0.9 has been chosen.
- number of served tasks (*nst*= 10000): finishing simulation parameter.
- probability of distributed task (*pdt*). Each generated task is a distributed one with probability *pdt*, and a local one with probability 1 - *pdt*. The density function is a bernoulli with a *pdt* Probability.
- maximum number of messages (*mnm*). For each distributed task, the number of receiving messages is generated. The density function is discrete uniformly in the interval [0 .. mnm]. The experimentation has been performed with a value of *mnm* = 5.

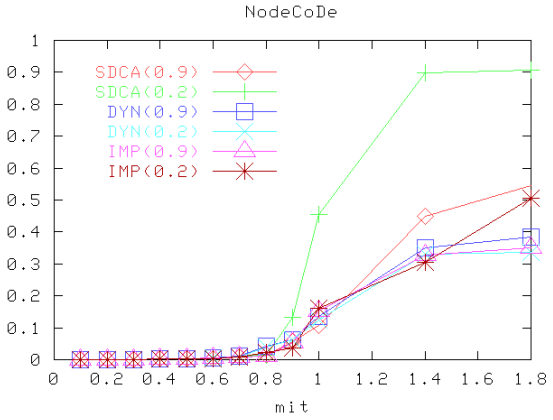


Figure 6: NodeCoDe. Numbers in parenthesis are *pdt* probabilities.

Generally, better results are obtained for large *mit* values and low *spin* and *share* parameters. For this reason, in the rest of the experimentation, values of *spin*=0.01 and *share*=0.1 have been chosen.

Fig. 6 shows the results for the NodeCoDe metric, defined as: relation between scheduled tasks with messages in MBQ and all the possible coscheduling ones (scheduled and not scheduled) into a node.

The good performance obtained for the SDCA model demonstrates its effectiveness in to profit the potential coscheduling. By extension, that result can be applied to each node of the overall system (or cluster), and for this reason no simulation for obtaining the overall cluster metrics has been performed.

6 Conclusions and Future Work

In a PVM environment made up of a NOW of Linux nodes, we have implemented and discussed different coscheduling techniques and compared their performance. Also, we have discussed their main advantages and drawbacks. A model, some related performance metrics and a coscheduling algorithm for dynamic coscheduling in Cluster computing is also proposed in this article.

Monito, a tool presented also in this article, serves for the analysis from the PVM queues, through the kernel queues, to the physical network device ones. This tool will allow in-depth study of the communication bottlenecks and correct these, for example the queue saturation cases shown in the experimentation.

Future work is directed towards investigating new dynamic coscheduling algorithms and implementing these jointly with the DCNDC in a real PVM and/or MPI Linux environment. Despite the presence of multi-processor nodes in the cluster, no consequences are produced in the model, as the model is only applied to the mono-processor ones. The future trend is to provide multi-processor capabilities.

Another goal is to expand *Monito* for also evaluating MPI communication performance and the design of new algorithms to decrease overhead in sampling data. Finally, we are interested in improving the performance of the communication system.

References

[1] Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems. In Third International Conference on Distributed Computing Systems, pp. 22-30. 1982.

[2] Arpaci, R.H., Dusseau, A.C., Vahdat, A.M., Liu, L.T., Anderson, T.E. and Patterson, D.A.: The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In Proceedings of the ACM SIGMETRICS'95/PERFORMANCE'95, pp. 267–278. 1995.

[3] Arpaci-Dusseau, A.C., Culler, D.E. and Mainwaring, A.M.: Scheduling with Implicit Information in Distributed Systems. In Proceedings of the ACM SIGMETRICS'98/PERFORMANCE'98. 1998.

[4] Dusseau, A.C., Arpaci, R. H. and Culler, D. E.: Effective Distributed Scheduling of Parallel Workloads. In Proceedings of the ACM SIGMETRICS'96. 1996.

[5] Sobalvarro, P.G., Weihl, W.E..: Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing, pp. 63–75. 1995.

[6] Sobalvarro, P.G., Pakin, S., Weihl, W. E., Chien, A. A.: Dynamic Coscheduling on Workstation Clusters. In Proceedings of the IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing. 1998.

[7] Wong, Frederick C., Arpaci-Dusseau, Andrea C., Culler, David E.: Building MPI for multi-programming systems using implicit information. In 6th European PVM/MPI User's Group Meeting. LNCS, Springer, pp. 215–222. 1999.

[8] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. MIT Press. 1994.

[9] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. www-unix.mcs.anl.gov/mpi. 1995.

[10] Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. www-unix.mcs.anl.gov/mpi. 1997.

[11] Buyya, R.: High Performance Cluster Computing: Architecture and Systems, Volume 1. Prentice Hall. 1999.

[12] Solsona, F., Giné, F., Hernández, P., Luque, E.: Synchronization methods in distributed processing. IASTED AI'99, pp. 471–473. 1999.

[13] Bailey, D. et al.: The NAS parallel benchmarks. International Journal of Supercomputer Applications. vol. 5 no. 3, pp. 63–73.1991.

[14] Kohl, J.A. and Geist, A.: XPVM 1.0 User's Guide". Technical Report ORNL/TM-12981, Computer Science and Mathematics Division, Oak Ridge National Laboratory. 1995.

[15] Yan, J.C., Schmidt, M. and Schulbach, C.: The Automated Instrumentation and Monitoring Systems (AIMS) - Version 3.2 User's Guide". NAS Technical Report NAS-97-001. 1997.

[16] Heath, M.T., Etheridge, J.A.: Visualizing performance of parallel programs. IEEE Software. vol 8 no. 5, pp. 29–39. 1991.

[17] Information Networks Division. HP Co.: Netperf: A Network Performance Benchmark. http://www.netperf.org/netperf/NetperfPage.html. 1996.

[18] Miller, B.P., Hollingsworth, J.K. and Callaghan, M.D.: Environments and Tools for Parallel Scientific Computing. J.J. Dongarra and B. Tourencheau (eds.), SIAM Press. 1994.