Implementing Explicit and Implicit Coscheduling in a PVM Environment*

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 with local workloads, balancing parallel efficiency against the local interactive response. Explicit and implicit coscheduling techniques in a PVM-Linux NOW (or cluster) has been implemented. Their performance and overheads executing local tasks and representative distributed benchmarks have been analyzed and compared.

1 Introduction

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 [5].

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.

Two-phase spin-block synchronization primitives used for *dynamic coscheduling*, named *implicit coscheduling* in [2], [3] and [4], only requires processes to block awaiting messages arrivals for coscheduling to happen. With two-phase spin-blocking, the waiting process spins for a determined time and if the response is received before the time expires then continues executing else the requesting process blocks and another one is scheduled.

Algorithms for implementing new explicit and implicit coscheduling environments are presented in this paper. Extensive performance analysis, as well as studies of the parameters and overheads involved in the implementation, demonstrated the applicability of the proposed algorithms in these new environments.

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

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1165–1170, 2000.

2 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 [5]). 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*):

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

Algorithm 1 *ImCoscheduling*. Implements the *implicit coscheduling*.

```
Initialize input_time, execution_time, sp
while (no_new_fragment) and (execution_time ¡ sp)
  and (execution_time ¡ timeout) do
      execution_time = current_time - input_time
if (no_new_fragment)
    if (timeout) then block (timeout - execution_time)
       else block (indefinitely)
```

Algorithm 2 *OneFragment*. Reads the fragment in only one phase.

```
call pvm_receive and wait until the fragment arrives
read the whole fragment ( header + body )
```

3 Algorithms

The implemented algorithms detailed in this section show how new distributed environments were created.

Function *ImCoscheduling* (Algorithm 1) implements implicit coscheduling by realizing a spin-block while the fragments (unit of PVM transmissions) composing a message are read. Algorithm 2, called *OneFragment,* reads each fragment in only one phase, instead of twice, as PVM does. Both algorithms were implemented in the *pvm_recv*( ) PVM routine.

Algorithm 3, *called Priority* was implemented outside the PVM, in a process named *Priority*, a copy of which is in each node of the cluster. It is responsible for assigning a high priority to distributed tasks. To do this only is necessary to assign a high priority (one unit level less than *Priority*) to *pvmd* in its creation.

4 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, 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 [6] were used: *is* and *mg*. In all the benchmarks, the communications between remote tasks was done through *RouteDirect* PVM mode.

4.1 Implemented Environments

The next distributed environments were created. The algorithm(s) used to implement each model is in parenthesis. PVM: original PVM. SPIN (1): the spin-block

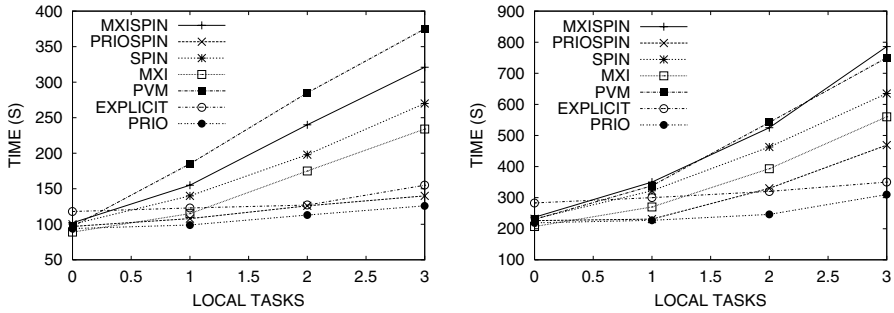

Fig. 1. *sintree* execution. (left) N = 30000. (right) N = 70000.

is only performed in the reading of the data fragment. MXI (2). MXISPIN (1, 2). PRIO (3). PRIOSPIN (1,3). 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.

Algorithm 3 *Priority.* Assigns a high priority to distributed tasks.

fork&exec (*pvmd*)
set priority (*pvmd* = max_priority - 1)

4.2 Results

Distributed Tasks Performance Fig. 1 shows the *sintree* execution times executing 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 tasks is the worst mode, by increasing the workload, its performance scarcely decreases due to T_{ex} does not vary. 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. 2 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.

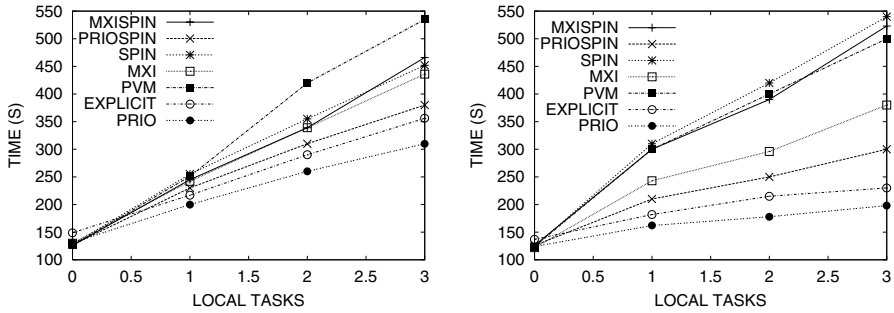

Fig. 2. Execution of the NAS parallel benchmarks (left) *mg* and (right) *is*.

Table 1. 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

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 1.

As might have been expected, when intensive message-passing distributed applications are executed (*sintree* and *is*), no effect in the local task is produced. 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 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. We are interested in developing a dynamical model and new coscheduling techniques for that environment.

References

1. Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems. In Third International Conference on Distributed Computing Systems. (1982) 22–30.

2. Arpaci, R.H., Dusseau, A.C., Vahdat, A.M., Liu, L.T., Anderson, T.E., Patterson, D.A.: The interaction of Parallel and Sequential Workloads on a Network of Workstations. SIGMETRICS'95. (1995). 267–278.
3. Arpaci, R.H., Dusseau, A.C., Culler, D.E., Mainwaring, A.M.: Scheduling with Implicit Information in Distributed Systems. SIGMETRICS'98. (1998).
4. Dusseau, A.C., Arpaci, R. H., Culler, D. E.: Effective Distributed Scheduling of Parallel Workloads. SIGMETRICS'96 . (1996).
5. Solsona, F., Giné, F., Hernández, P., Luque, E.: Synchronization methods in distributed processing. IASTED AI'99. (1999) 471–473.
6. Bailey, D. et al.: The NAS parallel benchmarks. International Journal of Supercomputer Applications. vol. 5 no. 3 (1991) 63–73.