A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations

Cosimo Anglano
Dipartimento di Scienze e Tecnologie Avanzate
Università del Piemonte Orientale
Alessandria (Italy)
email: mino@di.unito.it

Abstract
Implicit coscheduling strategies enable parallel applications to dynamically share the machines in a Network of Workstation (NOW) with interactive, CPU and I/O-bound sequential jobs. In this paper we present a simulation study that compares 12 coscheduling strategies in terms of their impact on the performance of parallel and sequential applications executed simultaneously on a NOW. Our results show that the coscheduling strategy has a strong impact on the performance of the applications (both parallel and sequential) composing the workload, and that no single strategy is able to effectively handle all workloads. In spite of that, our results can be used to identify the strategy that represents the best choice for a given application class, or the best compromise for various workloads. Moreover, we show that in many cases simple strategies outperform more complex ones.

1. Introduction

One of the factors that are contributing to the success of Networks of Workstations (NOWs) is the possibility of using them as general-purpose compute servers for the execution of parallel and sequential applications submitted by many competing users. To completely take advantage of this opportunity, however, effective scheduling techniques enabling parallel applications to dynamically share workstations with interactive, CPU-bound, and I/O-bound sequential processes, are needed.

For general purpose workloads, time-sharing approaches are attractive because they provide good response times for interactive jobs and good throughput for I/O-bound jobs. The most straightforward approach to time-sharing a NOW is to leave to each workstation the task of scheduling its processes (both sequential and parallel) independently from the other machines in the NOW. Unfortunately, this form of uncoordinated scheduling severely hurts parallel applications performance [1, 6]: satisfactory performance can be achieved only if communicating processes are simultaneously scheduled (coscheduled [14]) on the respective workstations. For instance, the completion time of a barrier synchronization is greatly reduced if participating processes arrive simultaneously at the barrier. By giving to processes the illusion of a dedicated cluster, coscheduling ensures that such a coordination is achieved, while uncoordinated scheduling cannot guarantee a good coordination among processes. A class of coscheduling strategies (that we call implicitly controlled or implicit for brevity) that have been recently proposed [6, 7, 13, 15] rely on local schedulers (to properly handle interactive and I/O-bound jobs mixes) and use the communication behavior of parallel processes to make scheduling decisions aimed at achieving a satisfactory approximation of coscheduling (of course, being the scheduling decision based on local knowledge, implicit coscheduling strategies may only be suboptimal). Compared with explicit coscheduling [2, 8, 14] (where the order in which communicating processes will be scheduled is determined a priori), these strategies are easier to implement on NOWs and scale better with the number of workstations in the cluster [6, 7, 13].

To use a NOW as a general-purpose compute server, the coscheduling strategy ensuring the best performance for all application classes of typical workloads should be adopted. However, given the disparate requirements and behaviors of the different classes, a strategy which is optimal for an application class may have detrimental effects on the other ones. It is therefore of crucial importance the characterization of each coscheduling strategy in terms of the performance it is able to guarantee to the various components of the workload.

In this paper we present a simulation study that compares 12 coscheduling strategies in terms of their impact on
the performance of parallel and sequential applications executed simultaneously on a NOW. We consider various simulation scenarios in which we vary the workload executed on the NOW and the scheduler used by the workstations. In particular, we consider workloads composed of (a) parallel applications only, (b) mixes of a parallel and many CPU-bound sequential applications, and (c) mixes of a parallel and many I/O-bound sequential applications. Moreover, we consider the schedulers of the System V Release 4 (SVR4) and the Sun Solaris operating systems. The two schedulers are nearly identical (Solaris is actually derived from SVR4) and differ only in the way they handle preemption: Solaris adopts immediate preemption (a running process is immediately preempted by a higher priority process [16]), while in SVR4 preemption is delayed until the expiration of the time quantum [17].

Compared with previous research [6, 13, 15], our work makes the following contributions. First, unlike previous studies, the implementations of the coscheduling strategies considered in this paper exactly match the original specifications. As a consequence, we show that strategies previously considered inadequate are instead able to deliver the best performance in many cases. Second, we consider more realistic workloads where parallel applications have to contend resources with both CPU-bound and I/O-bound sequential jobs. This enables us to characterize the behavior of the various strategies for scenarios not considered before and to identify the strategies that represent the best compromise for all the workloads mentioned before (or those that optimize performance for particular application classes). Third, we consider a larger set of coscheduling strategies, and we show that in some situations simple strategies (not considered in previous studies) may be as effective as more complex ones, with evident advantages in terms of implementation complexity. Finally, we consider two different operating system schedulers, and we show that their impact, while significant in terms of absolute performance, does not change the relative ranking among strategies for all but the most communication intensive workloads.

The paper is organized as follows. We start by reviewing the related work section in Section 2, and we continue with Section 3, where we present the coscheduling strategies considered in this paper. The experimental methodology followed in our work is described in Section 4, while Section 5 discusses the results we obtained from our experiments. Finally, Section 6 concludes the paper and outlines future research work.

2. Related Work

Despite the importance of the problem, at the best of our knowledge a complete characterization of coscheduling strategies has not been published in the literature. The most comprehensive study has been reported in [13], where 9 coscheduling strategies have been implemented and evaluated on a NOW of UltraSPARC workstations connected by Myrinet. Besides the exclusion of one of the possible message waiting actions (i.e., immediate blocking), and the introduction of some simplifications (w.r.t. the original specification) in the implementation of some strategies, the above study considered workloads made up by only parallel applications, and therefore did not consider the interactions of parallel and sequential jobs (both CPU and I/O bound), although one of the parallel applications communicated so little that the authors considered its processes as a set of sequential jobs.

In [15] only 4 coscheduling strategies have been implemented and compared, and in the various experiments only a single parallel application was executed in competition with a static set of sequential jobs, that is the same processes were run for the entire execution of the parallel applications. In real cases, however, this set is modified by the termination of some processes and by the arrival of new ones, and events of the latter type affect (as shown by our results) the behavior of the implicit coscheduling strategy, since newborn processes tend to have a priority higher than "old" ones.

In [6] only a subset of coscheduling strategies is analyzed via simulation for a variety of fictitious parallel applications; the presence of competing jobs is abstractly represented by using random duration times of the various activities carried out by the parallel processes.

Finally, in all the above studies the Solaris operating system has been assumed. However, as shown by our results, the results obtained for this operating system do not hold when operating systems with a different preemption policy (as, for instance, System V Release 4) are run on the workstations.

3. Coscheduling Strategies

As discussed in [13], an implicit coscheduling strategy tries to achieve process coordination by combining the effects of the message waiting action, taken by a receiving process, with those of the message handling action, performed by the operating system to handle incoming messages. As a matter of fact, a suitable waiting action (e.g. spin wait), may increase the chances that a receiving process is scheduled when the message arrives. If, conversely, the process has not executed yet the receive statement or is descheduled when the message arrives, an appropriate message handling action (e.g. preempting the running process) may help it to "catch up" with the sending process and to become coscheduled with it.

In this paper we consider a set of 12 coscheduling strategies obtained by combining four message waiting ac-
tions with three message handling actions. Table 1 reports the above coscheduling strategies, and for each strategy indicates the corresponding waiting and handling actions (e.g., DCS-IB is obtained by combining Immediate Boost with Block). In the above taxonomy, Implicit Coscheduling [6, 7] corresponds to SB, while Dynamic Coscheduling [15] corresponds to DCS.

The first two message waiting actions, namely Spin and Block, are the most straightforward ones. In the Spin case, a waiting process simply spins until the message arrives; coscheduling is achieved if the message arrives when the process is running. Conversely, in the Block case the process blocks immediately and is awakened by the operating system when the message eventually arrives; coscheduling may be achieved if the priority increase (priority boost) it receives from the system because of its block is sufficient to bring it on the CPU almost immediately. Spin-Block is a compromise between busy waiting and immediate blocking, and is based on the heuristics that a process waiting for a message should receive it within a reasonable amount of time if the sender is also scheduled currently; consequently, it spins for a certain amount of time and then blocks if the message is not received within that time. Since spinning is beneficial only if the time the process has to wait is smaller than the time \(B\) it takes to regain the CPU when it is awakened, the process computes the expected message arrival time, compares it with (an estimate of) \(B\), and decides whether it is better to spin for \(B\) time units or to block immediately [7]. To account for random fluctuations of the message arrival times due to the presence of competing jobs, we compute the 90\% confidence interval of the average message interarrival time, and we use the lower limit of the interval instead of its mid value. Note that \(B\) and the message arrival time are autonomously estimated by each process before each receive operation, so this strategy corresponds to adaptive two-phase waiting [11], since the spin time varies from operation to operation but is predetermined before the current operation begins, and is different from conditional two-phase waiting used in [7], and from fixed two-phase waiting [14] used in [13] and [15] for their studies. Finally, in Spin-Yield [13] if a waiting process decides to spin (as in Spin-Block) and the spin time elapses before the message arrives, it (a) lowers its priority (to a level that is one below the lowest priority of a parallel process of the same workstation) and (b) boosts the priority of one of the processes that have pending (i.e. unconsumed) messages.

Let us describe now the message handling actions. In the None case, no explicit action is taken upon message arrival. In the Immediate Boost strategy [15], the priority of the destination process, upon message arrival, is boosted to the maximum value and the currently running process is preempted if a fairness condition, ensuring that the process does not monopolize the CPU, is met. The fairness condition states that the priority boost is given if \(2^E(\Delta_T + C) \geq Q \cdot J\), where \(\Delta_T\) is the amount of time elapsed since the last priority boost, \(Q\) is the minimum length of the time slice, \(J\) is the number of jobs in the ready queue, and \(C\) and \(E\) are constants that depend on the workload. Finally, in the Periodic Boost strategy [13], the priority of receiver processes is not boosted immediately upon message arrival, but a kernel thread periodically inspects (in a round robin fashion) the message queues of parallel processes, and boosts the priority of the first process with a pending message. In both Immediate Boost and Periodic Boost, priority boost is performed even if the destination process has not executed the receive operation yet.

### 4. Experimental Methodology

Our study has been carried out by means of a simulator that models the schedulers of various operating systems, the cost of context switches, the overhead of communication operations, and the performance of the communication network. The simulator is driven by a set of synthetic applications (both sequential and parallel), whose computations are expressed by means of a simple language that allows the specification of CPU and I/O-based statements, as well as of the most common point-to-point and collective communication operations. The durations of these operations may be specified either as constants or random variables with different distributions.

We considered four synthetic parallel applications, directly derived from the NAS Parallel Benchmarks suite [3], namely CG, MG, LU, and IS. For each application we have kept unchanged the process structure, the communication topology, and the message sizes. The durations of the sequential portions of code have been measured by instrumenting the applications and executing them on a cluster of Pentium III/500 workstations. The number of iterations performed by each process has been reduced with respect to
the original applications in order to keep tractable the computational demand of the simulations. The characteristics of the various applications are summarized in Fig 2, where in the first column we indicate the communication pattern (the term Local indicates that each process communicates with a subset of other processes), in the second the percentage of the total execution time spent in communication operations, in the third the number of messages sent by each process, and in the fourth the message size distributions.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Comm.%</th>
<th>Msg.#</th>
<th>Msg. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Local</td>
<td>45%</td>
<td>707</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 byte: 57.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30KB: 14.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>110KB: 28.5%</td>
</tr>
<tr>
<td>LU</td>
<td>Local</td>
<td>5.5%</td>
<td>580</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40 bytes: 48.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>320 bytes: 48.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21 KB: 4.4%</td>
</tr>
<tr>
<td>MG</td>
<td>Local</td>
<td>41%</td>
<td>720</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8KB: 100%</td>
</tr>
<tr>
<td>IS</td>
<td>All-to-all</td>
<td>15%</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 byte: 50%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32 KB: 50%</td>
</tr>
</tbody>
</table>

Table 2. Characteristics of the parallel applications

Because the space covering the relevant parameters of our study is too large to explore exhaustively, we fix a number of system characteristics. In all the experiments, we have set the cluster size to 16 workstations, and for each workstation we have assumed a context switch time of 100 μs and an overhead for send/receive operations of 0.1 μs per byte. Two different schedulers, namely SVR4 and Solaris, have been considered in our experiments (in a given experiment, however, the same scheduler was used by all the workstations). Moreover, we have assumed that the communication network has a latency of 10 μs and a transfer rate of 100 MByte/s (values typical of modern light-weight messaging layers running atop gigabit switched LANs [9]). Finally, for Immediate Boost we have set C = 0 and E = 2, while for Periodic Boost we have used a period of 10 ms (as indicated in [13]) between consecutive sweeps of the message queues.

5. Experimental Results

In this section we present the results of our simulation experiments. We start by considering a workload composed of parallel applications only, and then we move to workloads composed of a mix of a parallel and many CPU-bound sequential applications. Finally, we consider workloads in which a parallel application is executed simultaneously to many I/O-bound sequential jobs. For all the results reported below, the confidence level and relative half width of the confidence intervals are 95% and 2.5% (or better), respectively.

5.1. Workload 1: Parallel Applications Only

Let us start with the results concerning a workload composed of the four parallel applications only. In our simulations, all parallel applications were started simultaneously. Fig. 1 reports the average slowdown of the various coscheduling strategies relative to ideal coscheduling, estimated as the ratio of the average completion time of the whole application set divided by the sum of the execution times of the various applications run in isolation (as discussed in [13], this corresponds to the ideal case of perfect time sharing). From Fig. 1, we observe that IB, DCS-IB, DCS-SB, and PB-IB are the best strategies for both the Solaris and the SVR4 schedulers (their slowdown is 1.07, i.e. they perform only 7% worse than ideal coscheduling). This result is in contrast with [13], where PB and DCS resulted the best strategies, and is due to the higher communication intensity of the applications considered in our study. For applications with high communication intensity, the best waiting action is indeed Immediate Blocking, since (a) the priority boost received upon message arrival is sufficient to bring almost immediately a blocked process on the CPU and (b) processes of other applications may progress in their computation. This explains also why coscheduling strategies based on Spin-Block (SB, DCS-SB, and PB-SB) have also very good performance. In the Solaris case, immediate preemption makes the blocking cost so low (practically identical to the context switch time) that the message interarrival time is very often smaller, so waiting processes block immediately practically always. In the SVR4 case their performance is slightly worse since the lack of immediate preemption results in a higher blocking cost, that sometimes makes receiving processes spin for a while before blocking. In these situations, if the message
arrives when the process is descheduled but it still has to complete the spin phase (i.e., its time quantum expired before the spin time elapsed), in the SB case its priority is not boosted (since it is not blocked), while in the PB-SB case it receives one or more priority boosts but may have to wait until the currently running process voluntarily relinquishes the CPU. Note that the poor performance observed in [13] for these strategies is due to the use of a constant, fixed spin time for all the receive operations, so waiting processes never blocked immediately (note also that in [13] Immediate-Blocking was not considered).

The fact that in DCS and PB waiting processes spin until a message arrives rather than blocking (either immediately or after a while) explains their very bad performance (note that they perform even worse than Local). In the Local case, the priority of a process that spins for all its quantum is lowered, so it will be scheduled again after the other competing processes have consumed a similar amount of CPU time. Conversely, the priority boosts given by DCS and PB raise the priority of processes even if they have already spent a large amount of time spinning, and the higher the communication intensity of a process, the higher the number of boosts it receives and, hence, its priority. Therefore, since communication intensive processes will be most of the time spin waiting for messages, they waste a large fraction of CPU time. It is worth to point out that if the communication intensity was lower (such as in [13]), the higher priority of these processes could be exploited to speed up the execution of the sequential portions of code between consecutive communication operations, improving consequently the performance of these strategies.

Finally, our results concerning SY match those reported in [13], but are significantly different for DCS-SY and PB-SY. This is explained by the fact that after the first yield, a process waits by just spinning (recall that in Spin-Yield a waiting process yields only once for a given message), so priority boosts have the same detrimental effect discussed for Spin. Furthermore, if more processes have pending messages, the one that gets the boost is chosen regardless of its communication intensity, since message queues are inspected in a round-robin fashion. However, to improve coscheduling, processes with higher communication intensities should receive a priority boost before processes with a lower intensity.

As a final consideration, we point out that our study (and, hence, our results) does not make any assumption about the behavior of parallel applications: their high communication intensities are due only to the speed of the processors and the communication network of the NOW considered here (higher than in [13]), and not to modifications of their structure or behavior.

5.2. Workload 2: Parallel applications and CPU-bound sequential jobs

Let us consider now the four workloads obtained by combining each parallel application with a stream of CPU-bound sequential jobs (background jobs). Background jobs arrive to each workstation according to a given arrival process, and each of them requires a random amount of CPU time. In all the experiments, background job interarrival times were hyperexponentially distributed with rate $\lambda = 0.08$ and coefficient of variation $c = 10$ (we have assumed a high variability of the process interarrival times as suggested in [4]), while their CPU demand was exponentially distributed with mean 5 sec. (comparable to the CPU time requested by parallel processes). Experiments for other values have been performed as well, but are not discussed here because of space constraints. In order to evaluate the impact of heavy-tailed distributions of process CPU times [5, 12], we have also performed experiments where the bounded Pareto distribution [10] has been used for the CPU demand, but we do not discuss them here because the results are not significantly different.

The average completion time of parallel and sequential applications are shown in Figs. 2 to 5 (note that the results for SY, DCS-SY, and PB-SY are omitted since for workloads with a single parallel application Spin-Yield behaves as Spin, so these strategies are identical to Local, DCS, and PB). From the above graphs, we see that for par-

![Figure 2. Average execution time of CG and CPU-bound jobs](image-url)

allel applications DCS-IB and DCS-SB result in the best performance for SVR4, while PB-IB and PB-SB are the best strategies in the Solaris case. The less effective strategies are, as expected, those based on pure Spin (i.e. Local, DCS, and PB), with the noticeable exception of PB for CG and MG. PB is the best strategy for both operating systems in the CG case (Fig. 2(a)), and for Solaris in the MG case (Fig. 3(a)); as discussed in Section 5.1, the priority
boosts increase the amount of spin time of communication-intensive processes and, hence, the probability that a waiting process is running when the message arrives.

The results concerning the performance of background jobs show instead that the worst strategies are those based on pure Spin, while the other ones are more or less equivalent. In general, the larger the communication intensity of the parallel application, the stronger the negative effect on the performance of CPU-bound jobs. Particularly evident is indeed the performance degradation for workloads that include CG and MG, as shown in Figures 2(b) and 3(b).

5.3. Workload 3: Parallel applications and I/O-bound sequential jobs

Let us consider now workloads composed of a single parallel application and a stream of I/O-bound sequential jobs, whose computation consists of a sequence of 100 iterations during which a CPU-burst is followed by an I/O-burst. The durations of CPU-bursts and I/O-bursts are exponentially distributed, with mean of 10 m/sec. and 20 m/sec., respectively. The distribution of interarrival times and its parameters are the same as in the CPU-bound case. The results shown in Figures 6 to 9 indicate that the best strategies for parallel applications are PB, PB-Ib, PB-Sb for both operating systems. This is not unexpected, since I/O-
performance are obtained for the SVR4 scheduler. For these applications, the number of messages exchanged by parallel processes is not sufficient to raise their average priority above that of I/O-bound jobs. Therefore, in the Solaris case they will be preempted very often by the I/O-bound jobs, that will be instead forced to wait the end of the current quantum to get the CPU in the SVR4 case.

The results for I/O-bound jobs are similar to those reported for CPU-bound applications. In particular, the worst strategies are those based on pure Spin, and the strategies that maximize the performance of parallel applications have the stronger negative impact on I/O-bound jobs.

5.4. Discussion

The results discussed in the previous Section show that no single coscheduling strategy is optimal for all the possible scenarios considered in this paper. It is however possible, with the help of Table 3, to identify the strategies that represent the best compromise for various workloads. This table summarizes the information concerning the relative performance of the various strategies (with the exception of those that have the worst performance for all the considered scenarios). In the Workload 1 columns, we have used the symbols +, -, and = to indicate that the strategy is the best, the worst, or stays in the middle, respectively. In the Workload 2 and Workload 3 columns, the symbols +, =, and - indicate instead that the strategy has the best performance for more than 2, for exactly 2, and for less than 2 parallel applications, respectively. If we rank coscheduling strategies in decreasing order w.r.t. the number of +, =, and -, we observe that:

- The strategies that represent the best compromise for all workloads are DCS-IB and DCS-SB in the SVR4 case (3 + and 2 = each), and PB-IB in the Solaris case (3 +, 1 =, and 1 -);

- For workloads composed of multiple parallel applications and CPU-bound jobs (columns Workload 1 and Workload 2), the best strategies are DCS-IB and DCS-SB in the SVR4 case (2 +, 1 = each), and IB in the Solaris case (2 +, 1 =);

- For workloads composed of multiple parallel applications and I/O-bound jobs (columns Workload 1 and Workload 3), the best strategies are DCS-IB, DCS-SB, and PB-IB in the SVR4 case (2 +, 1 = each), and PB-IB and PB-SB in the Solaris case (2 +, 1 = each).

6. Conclusions

In this paper we have presented a comparative evaluation of the various implicit coscheduling strategies proposed in the literature. As expected, our results show that the coscheduling strategy has a significant impact on the performance of the various application classes composing typical NOW workloads, and consequently a poor choice may result in unacceptable performance. Among the many factors that influence the performance that each strategy can deliver to the various applications classes, we have found that
<table>
<thead>
<tr>
<th>Workload 1</th>
<th>Workload 2</th>
<th>Workload 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Par.</td>
<td>Par.</td>
</tr>
<tr>
<td>SVR4</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Solaris</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SVR4</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Solaris</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IB</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DCS-IB</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DCS-SB</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PB-IB</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>PB-SB</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 3. Summary of relative performance of coscheduling strategies

particularly important is the ability to schedule a receiving process as soon as possible, as consequence of either an explicit action (e.g., immediate preemption) or of the workload behavior. In situations where this ability is present, a very simple strategy (IB) may outperform more complex ones (for both parallel and sequential applications) that use sophisticated heuristics to guide scheduling decisions, at least for workloads that do not include I/O-bound jobs. If, however, such jobs are present, more complex strategies are required in order to achieve satisfactory performance for parallel applications, although they have a detrimental effect on the performance of I/O-bound jobs. As a final consideration, we note that no single coscheduling strategy is able to effectively handle all workloads. The development of such a strategy remains an unsolved research problem that deserves further investigation.

Acknowledgments
The author wishes to thank Gianfranco Balbo for fruitful discussions and for helpful comments on the first version of this paper.

References


