Effective Distributed Scheduling of Parallel Workloads

Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler

Computer Science Division
University of California, Berkeley
{dusseau, remzi, culler}@CS.Berkeley.EDU

Abstract

We present a distributed algorithm for time-sharing parallel workloads that is competitive with coscheduling. *Implicit scheduling* allows each local scheduler in the system to make independent decisions that dynamically coordinate the scheduling of cooperating processes across processors. Of particular importance is the blocking algorithm which decides the action of a process waiting for a communication or synchronization event to complete. Through simulation of bulk-synchronous parallel applications, we find that a simple two-phase fixed-spin blocking algorithm performs well; a two-phase adaptive algorithm that gathers run-time data on barrier wait-times performs slightly better. Our results hold for a range of machine parameters and parallel program characteristics. These findings are in direct contrast to the literature that states explicit coscheduling is necessary for fine-grained programs. We show that the choice of the local scheduler is crucial, with a priority-based scheduler performing two to three times better than a round-robin scheduler. Overall, we find that the performance of implicit scheduling is near that of coscheduling (+/- 35%), without the requirement of explicit, global coordination.

1 Introduction

In this paper, we introduce *implicit scheduling*, a new distributed method for time-sharing parallel workloads. The philosophy of implicit scheduling is that communication and synchronization events within the parallel applications provide sufficient information for coordinating the scheduling of cooperating processes. By observing the interval that processes wait for communication and synchronization events to complete, each local scheduler is able to make independent decisions that tend to schedule the processes of a parallel application in a coordinated manner across processors.

We evaluate the performance of implicit scheduling relative to a zero-overhead implementation of *coscheduling* [35]. With coscheduling, processes of a parallel job are run at the same time across processors in an *explicit* manner: an externally controlled context-switch occurs simultaneously across all machines. The benefit of coscheduling is that cooperating processes behave as if they were running in a dedicated environment and can spin-wait during communication and synchronization.

Coscheduling is not well-suited to emerging processing environments and workloads. First, fault-tolerant, scalable coscheduling is non-trivial to design and implement [14]. Coscheduling has traditionally been used in tightly-coupled parallel processing environments that are viewed as a single system, where the failure model is the entire machine fails if one processor dies. This view is not practical in a distributed setting such as a network of workstations. Similarly, coscheduling is designed for workloads of only parallel jobs; as a result, it ignores the needs of mixed workloads containing interactive and I/O-intensive jobs. If interactive jobs are scheduled in the same round-robin manner as coscheduled parallel jobs, then users experience excessive delays. Preserving interactive response time requires raising the priority of interactive jobs; however, this perturbs the scheduling of parallel jobs, drastically reducing the performance of frequently communicating parallel applications [2, 4, 26]. Coscheduling also requires that processes busy-wait during I/O, wasting processor cycles and reducing throughput.

Local scheduling parallel jobs, where the existing operating system on each processor schedules processes independently, has a number of potential structural and performance advantages over coscheduling. First, local scheduling exists on every machine, requiring no additional implementation. Since each scheduler is independent, the system trivially provides fault-containment: a parallel job running on a subset of machines is not affected by the failure of others [6]. Second, time-sharing priority-based local schedulers have been carefully tuned for both interactive and I/O-intensive processes [13, 18, 25]. To improve throughput, jobs waiting on I/O relinquish the processor; to improve response time, jobs that block frequently are given higher priority when they have computation. For these reasons, we are motivated to investigate whether or not local schedulers can be used to effectively schedule parallel workloads.

Several previous researchers have compared the performance of coscheduling to approaches using the local schedulers, and have unanimously concluded that local scheduling is insufficient for fine-grained parallel applications [2, 8, 15, 16, 19]. The poor performance of local scheduling occurs because cooperating processes are not scheduled simultaneously across processors; as a result, when a process attempts to communicate or synchronize with another process, the other may not be scheduled. Thus, the programmer or run-

[0]This work is supported in part by the Advanced Research Project Agency (F30602-95-C-0014), the National Science Foundation (CDA 94-01156 and CCR-9210260), and the California State MICRO Program. Dusseau is also supported by an Intel Foundation Graduate Fellowship.

SIGMETRICS 96-5/96 Philadelphia, PA, USA

time system must choose an action after initiating a communication request: spin-wait until the response is received or block and switch to a competing job. If a process spin-waits for the response, processor time is wasted while the process spins idly; alternatively, if the process blocks, time is wasted context-switching and thrashing between competing processes.

In order to use local schedulers for parallel jobs, communicating processes must be dynamically identified and coordinated. The component of implicit scheduling that achieves this coordination is *two-phase* blocking. With two-phase fixed-spin blocking, the waiting process spins for some predetermined time and, if the response is received before the time expires, continues executing; however, if the response is not received, the requesting process blocks and another process is scheduled. Unfortunately, selecting the spin-time for events such as barriers in a multiprogrammed environment is non-trivial [43]. An *adaptive* two-phase algorithm that adjusts spin-time according to run-time measurements of waiting intervals is beneficial under these conditions.

In this paper, we show that implicit scheduling is a feasible alternative to coscheduling, given a realistic bulk-synchronous programming model, a commercial priority-based local scheduler, and an adaptive two-phase blocking algorithm. This combination enables processes to influence the collection of local schedulers and "coordinate" themselves during communication phases, while remaining independent during computation phases. Our results hold for fine- and coarse-grained parallel applications, programs with varying degrees of load-imbalance, and for various communication patterns. We also show that our conclusion is robust to changes in network performance and context-switch costs.

The rest of this paper is organized as follows. We begin in Section 2 by reviewing the research in parallel scheduling. In Section 3 we describe our parallel workload and our simulation environment. The results in Section 4 examine the performance of local scheduling when processes block immediately after communication events. In Section 5 we explore the impact of two-phase fixed-spin blocking. We present two-phase adaptive blocking algorithms in Section 6 and demonstrate how barrier operations within the parallel application can collect data for the scheduler at run-time. Section 7 examines the sensitivity of implicit scheduling to the underlying local scheduler. Finally, we present our conclusions in Section 8.

2 Related Work

The scheduling of parallel jobs has long been an active area of research. It is a challenging problem because the performance and applicability of a parallel scheduling algorithm is highly dependent upon a host of factors at many different levels: the workload, the parallel programming language, the blocking algorithm, the local operating system, and the machine architecture.

Parallel scheduling is composed of at least two interdependent steps: the allocation of processes to processors (space-sharing) and the scheduling of the processes over time (time-sharing). A large number of studies have focused on the processor allocation step of parallel job scheduling [5, 7, 17, 19, 27, 29, 30, 31, 33, 36, 37]; fewer have investigated the second step. We believe that a mixed approach, utilizing both space-sharing and time-sharing, is required to maintain interactive response times and high throughput. In this paper, we consider the problem after jobs have been allocated to a set of processors and focus on how to time-share between competing jobs on the same processor.

In 1982, Ousterhout introduced the idea of *coscheduling* [35], in which cooperating processes from a single job are simultaneously scheduled across processors, giving each job the impression that it is running on a dedicated machine. With coscheduling, the destination process of a message is guaranteed to be scheduled; therefore, synchronization time is the same as in a dedicated environment and processes can spin while waiting for communication and synchronization responses.

Previous work comparing the performance of coscheduling to local scheduling has unanimously concluded that explicit coscheduling is necessary for fine-grained parallel applications, since local scheduling did not coordinate the communication phases across processes [2, 8, 15]. However, many of these studies did not thoroughly investigate the impact of different blocking algorithms, local schedulers, or programming models. Most studies have considered only spin-waiting or immediate blocking; those considering two-phase blocking have not thoroughly investigated the effect of different fixed-spin times. Furthermore, most previous work has considered simplified local schedulers, such as round-robin, rather than the priority-based schedulers used in commercial systems.

In a paper exploring the merits of runtime identification of cooperating processes [16], the performance of local scheduling with two-phase blocking is examined. However, little improvement is seen due to the use of a short spin-time relative to the context-switch cost. In an environment much different than ours, the effect of different fixed-spin times is quantified for shared-memory lock synchronization [19]; with local scheduling, they find spinning for the context-switch cost gives the higher utilization than other fixed-spin times, but that coscheduling results in the best utilization.

A small number of studies have examined the interaction between multiprogramming and adaptive two-phase synchronization algorithms. Karlin et. al. [22] show that several adaptive algorithms perform better than fixed-spin algorithms for shared-memory lock synchronization. However, appropriate adaptive blocking techniques for barriers are drastically different than those for locks, due to the global cost of blocking at barriers. Previous research applying adaptive algorithms to barrier synchronization has assumed a radically different programming model from ours, in which only threads from the same parallel job may execute on the same processor [24, 42].

Finally, in another time-sharing approach currently under development, communicating processes are dynamically coscheduled by using message arrivals to trigger scheduling events [39]. However, the authors do not explore algorithms for releasing the processor on the sending side. We believe that their approach could be complementary to ours and plan on investigating this in future work.

3 Methodology

In this section, we give an overview of the experimental method used in this study. We begin by presenting a realistic model of parallel programs, which we use for the synthetic applications in our simulations. We then describe our event-driven simulator and the details of the local scheduler. We conclude by examining the execution characteristics of our synthetic workload in a coscheduled environment.

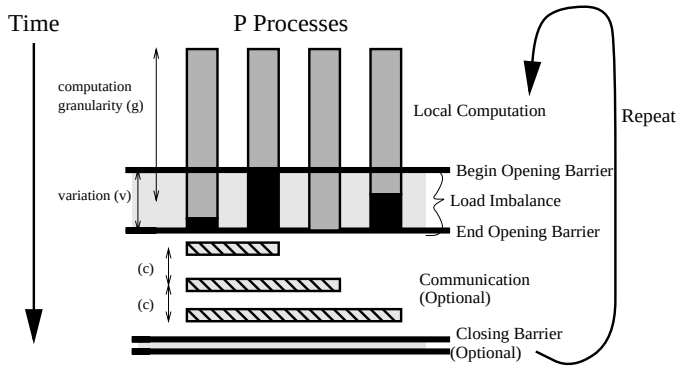

Figure 1: *Bulk-Synchronous SPMD Model of Synthetic Parallel Applications.* Each process of a parallel job executes on a separate processor and alternates between computation and communication phases. Processes compute for a mean time g before executing the opening barrier of the communication phase; the variation in computation across processes is uniformly distributed in the interval $(0, v)$. Within the communication phase, each process computes for a small time, c, between events, and the phase may close with a barrier.

3.1 Programming Model

We use the popular single-program multiple-data (SPMD) programming model, where a parallel *job* consists of one *process* per processor on a static number of processors throughout execution. It has been our experience, as well as the experience of others [10, 11, 12, 20, 38], that the structure of many parallel applications is *bulk-synchronous*; *i.e.*, phases of purely local computation alternate with phases of interprocessor communication. Figure 1 demonstrates the programming model that we use in this study. Each of P cooperating processes in a parallel job executes on a separate processor. Each process computes locally for a time selected from a uniform distribution in the interval $(g - \frac{v}{2}, g + \frac{v}{2})$. By adjusting g we model parallel programs with different computation granularities and by varying v we change the load-imbalance across processors. The communication phase consists of an *opening barrier*, followed by an optional sequence of pairwise communication events separated by small amounts of local computation, c, and finally an optional *closing barrier*.[1] We vary the patterns within the communication phase to adjust the degree of inter-process dependencies. After completing the communication phase, each process begins the local computation phase again.

The flexibility of the communication events allows us to capture a rich variety of programming models. For example, the pairwise communication events can be synchronous or asynchronous sends and receives on a distributed-memory machine or remote reads and writes in a shared address space. In this study, we focus on *read* operations, such as those built on top of active messages [41] or cooperative-shared memory [21]. A read operation is a request-response message pair, constructed by sending a request to the processor containing the desired memory location; the destination process reads the local data and sends the response back to the requester. In our model, the read request can be serviced only when the process from the same job is scheduled. Clearly, if processes are not scheduled simultaneously across processors, then the requester may wait an unpredictable amount of time for the read response.

[1] Our barrier is implemented assuming no hardware support. A process reaching the barrier sends a message to the *root* process. When the root handles the message, it increments a counter; once the counter reaches P, the root broadcasts a barrier-completion message. The broadcast is implemented using P pair-wise messages instead of a tree-based structure; otherwise, processes at the tree leaves would need to wait for processes at intermediate nodes to be scheduled. Note that because we do not currently model the overhead of sending messages, the broadcast is free to the root process.

3.2 Simulation Parameters

We have developed a simulated environment to study the behavior of synthetic parallel applications under different scheduling approaches and with new blocking algorithms. We are able to explore the impact of conditions such as the characteristics of the parallel applications, the performance of the communication network, and the cost of a context-switch.

Our simulator is driven by synthetic parallel applications adhering to the model in Figure 1. Each parallel job is specified by an arrival time and the number of processes in the job (P). The communication behavior of a job is defined by the mean computation time between communication phases (g), load-imbalance (v), the time between reads within a communication phase (c), the number of communication phases to execute before termination, and the communication pattern.

We investigate three different communication patterns: BARRIER, NEWS, and TRANSPOSE. The BARRIER pattern consists of only the opening barrier, and thus contains no additional dependencies across processes; the other two patterns consist of a sequence of reads surrounded by an opening and a closing barrier. The communication phase of TRANSPOSE contains P reads, where on the i-th read, process p reads data from process $(p+i) \bmod P$; therefore, each process depends on all other processes. In NEWS, a grid communication pattern is used; each process depends upon its four neighbors.

Within the communication layer, the simulator can model the effects of packet sizes and limited message buffering, but due to time constraints, we do not examine those effects in our experiments. For simplicity, each read requests only a single packet of data. We do investigate the impact of network latency (L), where latency is defined as the total time to deliver one packet from the source to the destination processor.

3.3 Local Scheduling

The policies of the local scheduler have a significant impact on the performance of parallel applications. Thus, to obtain realistic results, we constructed the local scheduling component of the simulator to closely match that of the Unix System V Release 4 (SVR4) [18] scheduler in both functionality and structure; in fact, significant portions of our code are copied directly from Solaris 2.4 source.

The objective of a time-sharing scheduler is to schedule each process fairly and efficiently. This goal is realized in SVR4 with a dynamic priority allocation scheme. A job's priority is lowered if it runs for its time quanta without relinquishing the processor; the job's priority is raised if it sleeps frequently. The SVR4 scheduler maintains fairness with a starvation timer that runs once every second; if a job has not completed its time quantum before the starvation timer expires, then its priority is raised. The new priority of

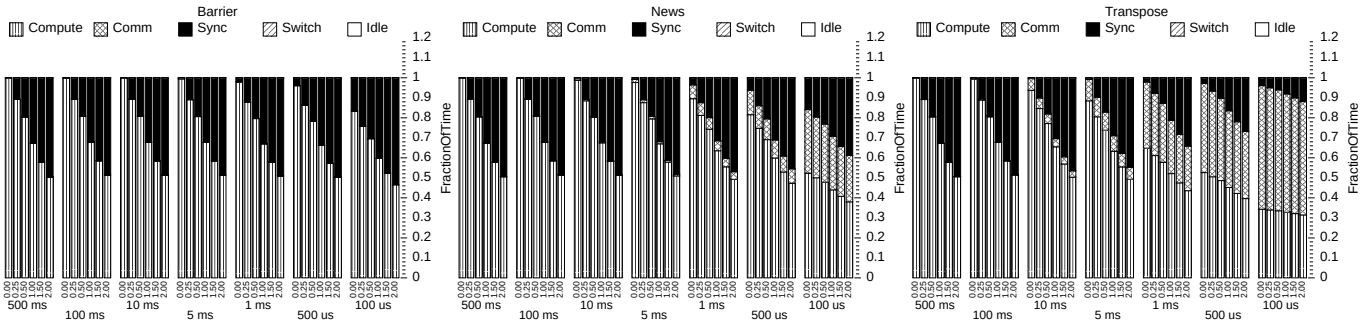

Figure 2: *Execution Characteristics as a Function of Computation Granularity and Load-Imbalance when Coscheduled.* The three graphs show the breakdown of execution time for the three communication patterns. Along the x axis, the computation granularity, g, and the load-imbalance, v, are varied. Each group of six bars has the same computation granularity. Groups on the left are relatively coarse-grained; groups on the right are more fine-grained. Within each group, the load-imbalance is increased as a function of the granularity: from $v = 0$ to $v = 2 \cdot g$. The fraction of execution time is divided into five components (computing, communicating, synchronizing, context-switching, and idling), ordered from bottom-to-top to match the left-to-right ordering in the key. Note that the BARRIER program performs no communication, and that all programs exhibit negligible context-switch and idle time. This data assumes a network latency of 10 μs.

a job (due to time quantum expiration, blocking, or starvation) and the number of clock ticks in a time quantum are specified in a user-tunable table.

We use the same mechanism to accommodate jobs sleeping on a communication or synchronization event as any other event (*e.g.*, disk I/O). When a job sleeps on an event, it is placed on a list of blocked jobs and will not be scheduled. When a message arrives for the blocked job, the job is temporarily given kernel priority; at this point, the job most likely has the highest priority on the processor, is scheduled, and handles the message. If handling the message unblocks the waiting job (*e.g.*, the job is waiting for a read response, which has now arrived), then the job is given a new user-level priority and placed on the run queue. If the job is still blocked (*e.g.*, the message is not a response, but a request from another job), then it is returned to the blocked list.

We make several *pessimistic* assumptions about the synchronization of events across processors. First, both the 10 ms clock tick and the one second update timer expire independently across processors (see Section 7 for the impact of simultaneous clock ticks). Second, if multiple parallel jobs arrive in the system at the same time, then the processes are randomly ordered in the local scheduling queues. Conversely, we make *optimistic* assumptions about the behavior of coscheduling: there is no skew of time quanta across processors and, more significantly, the global context-switch cost is identical to the local scheduler cost.

3.4 Characteristics of Synthetic Workload

Because the space covering the parameters we have introduced is too large to explore exhaustively, we fix a number of the workload and system characteristics. We consider three parallel jobs with the same computation granularity, load-imbalance, and communication pattern arriving at the same time in the system. We fix the communication granularity, c, at 8 μs. The number of computation/communication iterations is scaled so that each job runs for approximately 10 seconds in a dedicated environment. The system consists of 32 available processors and each job requires 32 processes.

To illustrate the communication and synchronization characteristics of our synthetic benchmark programs as a function of the communication pattern, granularity, and load-imbalance, Figure 2 shows the fraction of execution time spent in one of five states when the programs are coscheduled.[2] Each bar shows the percentage of time spent in one of the following states averaged over all processors: computing, communicating, synchronizing, context-switching, and idling. The communicating and synchronizing costs are the time spent spinning while waiting for an event to complete. The processor is idle when all of the processes are blocked.

Examining the breakdowns of each bar, we see several trends as the communication characteristics of the programs are varied. First, as the load-imbalance in the programs increases (*i.e.*, moving to the right *within* a group of six bars), the time spent spinning during synchronizing increases. Second, for the NEWS and TRANSPOSE programs, as the computation granularity decreases (*i.e.*, moving to the right *across* groups of bars), the communication-to-computation ratio of each program increases. We can also see that our coscheduled environment is well-tuned; *i.e.*, no idle time exists and the time spent context-switching is negligible, due to the long time quantum (500 ms) relative to the context-switch cost (200 μs).

4 Local Scheduling with Immediate Blocking

In this section, we verify previous studies [8, 15] that showed coscheduling is superior to local scheduling with immediate blocking for fine-grained parallel jobs. We extend these results by investigating performance over a range of communication patterns and machine parameters, in order to identify the ranges which require improvement.

4.1 Verification of Previous Results

The graphs in Figure 3 show the slowdown of local scheduling with immediate blocking relative to coscheduling as a function of computation granularity and load-imbalance for three competing jobs. The context-switch time is fixed at 200 μs for both local scheduling and coscheduling; the communication latency is set to 10 μs. The breakdowns of each bar show the percentage of time spent in each of the five

[2] The data presented in the figures of this paper can be found on the World Wide Web at http://now.cs.berkeley.edu.

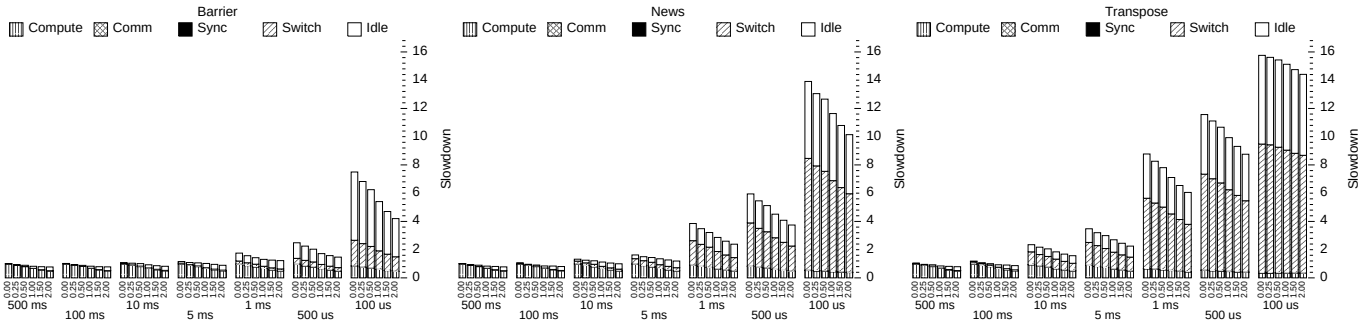

Figure 3: *Local Scheduling with Immediate Blocking.* The metric along the y axis is the slowdown of the workload with immediate blocking and local scheduling versus coscheduling. Blocking at every communication and synchronization point is effective when the communication granularity is large due to load-imbalance effects, but terribly costly for fine-grained programs. Note the large switch and idle time, but no time spent spinning on synchronization or communication events. Latency is fixed at 10 μs and context-switch cost at 200 μs.

Barrier			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
			Low	High	Low	High	Low	High
Context-Switch	Low	Latency Low	1.05	1.01	0.97	0.82	0.93	0.80
		Latency High	2.29	1.62	1.01	0.83	0.92	0.82
	High	Latency Low	2.89	2.43	1.07	0.95	0.93	0.81
		Latency High	6.83	4.70	1.09	0.96	0.94	0.80

News			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
			Low	High	Low	High	Low	High
Context-Switch	Low	Latency Low	0.96	0.96	1.01	0.88	0.91	0.81
		Latency High	3.64	3.08	1.11	0.94	0.93	0.79
	High	Latency Low	3.07	2.93	1.36	1.18	0.93	0.80
		Latency High	13.1	10.8	1.50	1.27	0.93	0.81

Transpose			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
			Low	High	Low	High	Low	High
Context-Switch	Low	Latency Low	0.82	0.82	0.91	0.87	0.92	0.82
		Latency High	4.31	4.07	1.50	1.24	0.94	0.80
	High	Latency Low	2.46	2.42	1.89	1.63	0.94	0.79
		Latency High	15.6	14.7	3.20	2.45	0.96	0.82

Figure 4: *Impact of Workload and System Parameters on Immediate Blocking.* Each entry in the table shows the slowdown of local scheduling with immediate blocking versus coscheduling as a function of four variables: two workload characteristics (columns) and two system parameters (rows). A fine-, medium-, and coarse-grained program is evaluated, where the computation granularity is 100 μs, 500 μs, and 500 ms, respectively; the load-imbalance is set to 0.25 and 1.5 times the granularity, for low and high values. For the system parameters, we consider low and high network latencies of 10 μs and 100 μs and low and high context-switch costs of 50 μs and 200 μs. **Black** regions indicate areas for which the distributed scheduling algorithm is at least 15% slower than coscheduling; the regions where the distributed algorithm are 15% better are **gray**.

phases in the locally scheduled workload. Our performance metric, slowdown, is the ratio of the workload completion time under local scheduling to the completion time under coscheduling.

Our results indicate that immediate blocking gives superior performance for coarse and medium-grain computations (5 ms to 500 ms) coupled with high load-imbalance (one to two times the granularity). These results are well understood: when the load-imbalance is large, blocking immediately is beneficial compared to coscheduling because the processor can switch to and execute another job instead of spinning uselessly.

At the other extreme, coscheduling is strictly superior for fine-grained jobs regardless of the load-imbalance. When the load-imbalance is small, blocking immediately is not appropriate because the cost of the context-switch is greater than the load-imbalance. Communication-intensive patterns, such as TRANSPOSE, are especially sensitive, as indicated by the increase in the time spent context-switching.

4.2 Sensitivity to Machine Parameters

The previous subsection assumed a fixed context-switch cost (200 μs) and communication latency (10 μs). However, recent measurements indicate that both of these costs vary widely in current systems. Table 1 shows that latencies vary between 2 and 100 μs in current MPP and workstation networks. Similarly, context-switch costs can vary between 10 and 200 μs, depending upon whether or not cache effects are taken into account [32]. Modeling the cost of a context-switch to the application requires considering the impact of memory; therefore, we examine a moderate and a high context-switch cost: 50 and 200 μs, respectively.

Figure 4 reports the slowdown of local scheduling with immediate blocking versus coscheduling over a set of realistic context-switch costs, network latencies, computation granularities, and load-imbalances. The figure clearly shows that the relative performance of the two scheduling approaches is dependent not only on the characteristics of the workload, but on the system parameters as well.

With a high latency network (100 μs) and a low context-switch cost (50 μs), local scheduling with immediate blocking outperforms coscheduling, due to its ability to overlap communication with computation. As long as a sufficient number of jobs exist in the system and the context-switch cost is much less than the round-trip time of the network, the processor can switch to another process and keep busy while the message travels in the network. When jobs are coscheduled, increasing network latency simply increases the

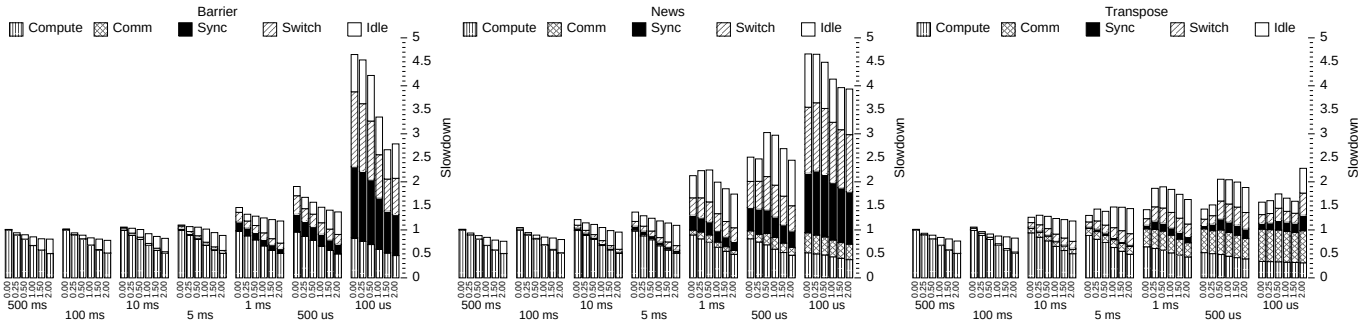

Figure 5: *Fixed-Spin Equal to Context-Switch Cost.* The slowdown of the workload is shown when local scheduling with a two-phase fixed-spin is used instead of coscheduling. Each process spins for the context-switch time (200 μs) before blocking on its communication and synchronization events.

System	Latency (μs)	Source
Cray T3D	2	[1]
TMC CM-5	6	[41]
Intel Paragon	6	[9]
FDDI	6	[28]
Myrinet	11	[9]
Fore ATM	33	[40]
Switched Ethernet	52	[23]
LattisCell ATM	104	[23]

Table 1: *Latency on Current Networks.* Measured latencies on modern MPP and workstation networks.

amount of time processes spin-wait for communication to complete. Few context-switches occur when processes are coscheduled; therefore, little benefit is gained by reducing their cost. An interesting implication is that optimizing the cost of the local context-switch improves the relative performance of distributed scheduling algorithms.

Figure 4 also identifies the performance regimes where the performance of the distributed algorithm must drastically improve to match coscheduling: medium- to fine-grained programs, except when the network latency is high and the context-switch cost is low. Since we are interested in environments conducive to parallel computing, low latency networks are required; a high context-switch cost appears to be a realistic, yet pessimistic, assumption for our environment. Therefore, in the experiments that follow, we concentrate on the regime where local scheduling needs the most improvement relative to coscheduling: low latency networks (10 μs) coupled with a high context-switch cost (200 μs).

5 Static Blocking Algorithms

In the previous section we saw that local scheduling with immediate blocking at synchronization and communication points leads to unacceptable performance for fine-grained parallel applications. We now investigate the impact of *two-phase fixed-spin* blocking mechanisms. In a two-phase fixed-spin algorithm, the waiting process spins for a predetermined time and, if the response is received before the time expires, continues executing; otherwise, the process blocks and another process is scheduled. We will see that spinning for at least twice the context-switch time dynamically coordinates communicating processes, resulting in performance near that of coscheduling.

5.1 Two-Phase Fixed-Spin

Previous research has shown that two-phase fixed-spin algorithms with a spin-time equal to the context-switch cost is beneficial when acquiring *shared-memory locks* [19, 22]. The intuition is that spinning is advantageous when waiting for a lock that will be released soon, since spin-waiting saves the processor the cost of switching to another process. However, if the wait time is long, blocking and switching to another process accomplishes more work. Our goal is to investigate how a two-phase fixed-spin algorithm performs when applied to the communication and synchronization events in our programming environment.

Figure 5 shows the performance of local scheduling with a fixed-spin time equal to the context-switch cost (200 μs) for reads and barriers. For coarse-grained programs with high load-imbalance, fixed-spin performs similarly to blocking immediately (shown in Figure 3), because processes reaching a barrier eventually block and switch to another process and the spin overhead is small relative to the total execution time.

For fine-grained computations, the performance when spinning for the context-switch time improves significantly compared to immediate blocking (from nearly 16 times slower than coscheduling, to less than five times slower). This improvement occurs because cooperating processes become dynamically coordinated after executing a barrier, as a side-effect of the SVR4-scheduler's policy of raising the priority of a process returning from sleeping on an event (described in Section 3.3). Coordinated scheduling is particularly beneficial after a barrier operation because the barrier precedes the communication phase; as a result, processes are able to read from one another and spin-wait on the response. With the definition that an event is *successful* if the process is still spinning when the event completes, 99% of the reads are now successful and less time is wasted context-switching.

The details of how the dynamic coordination occurs after processes execute a barrier are as follows. As described in Section 3.1, when the last process arrives at the barrier, the other processes are notified; at this point there are two possible scenarios: the notified process is still spin-waiting for the barrier to complete, or, in the case that the process blocked on the barrier, another process is running. If the process is blocked, the local scheduler awakens it and gives it a new priority in the upper range of the user-level pri-

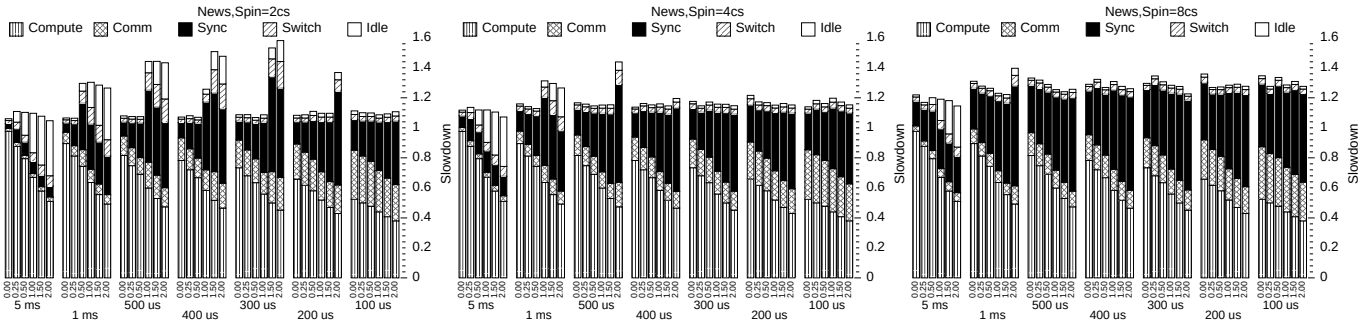

Figure 6: *Varying the Fixed-Spin Time.* The slowdown is shown when local scheduling with different fixed-spin lengths is used compared to coscheduling. The amount of time each process spins before blocking on communication and synchronization events is varied for the NEWS pattern. The fixed-spin amounts are two, four, and eight times the context-switch cost. Note that the x axis in this graph focuses on finer granularities (5 ms to 100 μs instead of 500 ms to 100 μs).

orities. At this higher priority, it is likely that the process returning from the barrier is the highest priority job on its processor. Since this raising of priorities and rescheduling occurs concurrently across processors, the cooperating processes are scheduled simultaneously across processors.

The improvement in performance of the TRANSPOSE pattern relative to the other communication patterns (*i.e.*, TRANSPOSE experiences slowdowns less than 2.5 relative to coscheduling, rather than slowdowns of 5) is due to the communication dependencies across processes. In TRANSPOSE, each process reads from every one of the other cooperating processes; the implication is that if all cooperating processes are not scheduled, then the processes will block on a read before reaching the closing barrier. However, if the processes successfully execute each read, then they also successfully complete the closing barrier and their scheduling remains coordinated. Conversely, in the NEWS pattern, a process can successfully complete its four read operations even when few other processes are scheduled, and block on the closing barrier; the result is that the processes do not become coordinated over time.

5.2 Varying Spin-Time

Two-phase blocking provides the basic mechanism behind implicit scheduling: processes are able to observe the behavior of their communication and synchronization events and to share this information with the scheduler. When a process times-out waiting for a communication event to complete, the implication is that the cooperating processes of this job are not currently scheduled; subsequently, the correct behavior for the process is to block and allow another process to be scheduled. However, if the communication event completes relatively quickly, this indicates that the processes are coordinated and should remain scheduled.

Our next step for improving the performance of our implicit scheduler is to examine the impact of different fixed-spin times on performance. We focus our next few experiments on NEWS because it appears to be the most difficult communication pattern to dynamically coordinate, but we return to the full set of patterns for our final simulations.

The left-most graph of Figure 6 shows that the performance of NEWS for fine-grained computations (note the change in the x axis) improves substantially when the spin-time is increased to twice the context-switch cost (400 μs). The improvement from 5 times slower than coscheduling down to only 60% slower is closely tied to the characteristics

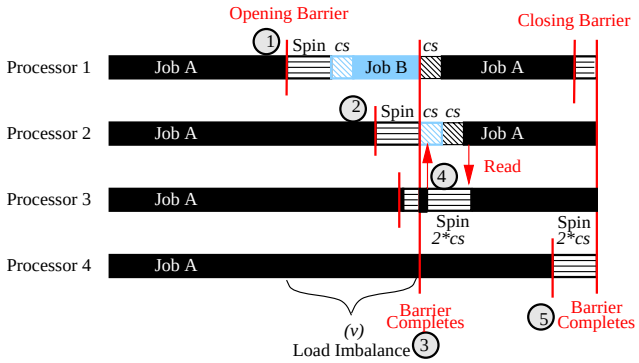

Figure 7: *Maintaining Dynamic Coordination.* To maintain the dynamic scheduling coordination that occurs after a barrier, processes at reads and barriers must spin for the amount of skew introduced by distributed scheduling. In this example, processor 2 begins switching to job B at the same time the other processors begin to execute job A; thus, two context-switches are required for processor 2 to begin responding to requests for job A.

of the local scheduler. Blocking on a barrier introduces a scheduling skew for cooperating processes across processors of twice the context-switch cost; if processes spin for this at least this amount before blocking, then they effectively smooth over the scheduling irregularities.

The cause of this scheduling skew is illustrated in Figure 7. Two competing parallel jobs, A and B, are allocated to four processors; job A is initially scheduled simultaneously across all four. The process on processor 1, A_1, is the first to reach the barrier (indicated by event 1 in the figure). A_1 spins for time $2 \cdot cs$ and then blocks, because the other processes have not yet reached the barrier; the processor then context-switches to job B. Meanwhile (at event 2), on processor 2, process A_2 arrives at the barrier, spins, and then blocks. However, shortly after process A_2 blocks (event 3), the last process, A_4, reaches the barrier and the processes of job A are notified. Each local scheduler now determines that A should be scheduled, but processor 2 requires $2 \cdot cs$ to switch to A. As a result, A_3 must wait at least twice the context-switch cost before receiving a response to its read-request (event 4) and all processes must also spin for $2 \cdot cs$

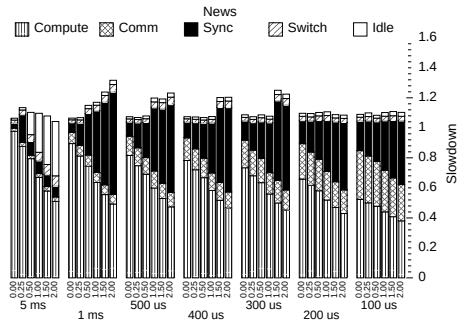

Figure 8: *Adaptive Two-Phase Blocking with Known Load-Imbalance.* Performance is shown for local scheduling with adaptive blocking when the system is aware of the load-imbalance in the programs. The slowdown is relative to coscheduling.

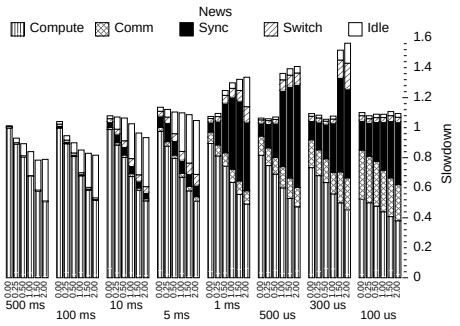

Figure 9: *Local Approximation of Program Load-Imbalance.* Local scheduling with a local approximation of load-imbalance performs worse than the oracle, due to low estimates of the imbalance. The slowdown is relative to coscheduling.

before the closing barrier completes (event 5). [3]

Other interesting points in the left-most graph of Figure 6 are the spikes of greater slowdown that occur when the load-imbalance is slightly greater than the spin-time of 400 μs. The decrease in performance occurs because processes do not spin long enough to wait through the load-imbalance across processes and, thus, few opening barriers complete successfully (*e.g.*, 37% are successful when the computation granularity is 400 μs and the load-imbalance is 800 μs (*i.e.*, $v = 2g$), versus 98% when the load-imbalance is only 200 μs (*i.e.*, $v = 0.5g$). The slowdown indicates that, due to losing the coordinated scheduling across processes, the cost of scheduling another process is much greater than the time spent context-switching. For this reason, it may be beneficial to spin longer before switching to another process.

The next two graphs of Figure 6 show that as the spin time is increased, processes remain coordinated during larger load-imbalance and the performance of those data points improves as expected. However, not only are there still load-imbalances for which processes lose coordination, but the slowdown of programs with lower load-imbalances increases. This slowdown occurs because the penalty when a process spins and then blocks on a barrier increases (*i.e.*, the spin-time has increased), while percentage of barriers that complete successfully remains constant near 98%. This is shown in the graph as an increase in the communication and synchronization portions of the bar graph.

6 Adaptive Blocking Algorithms

In the previous section, we examined two-phase fixed-spin algorithms and saw that the performance of applications is sensitive to the amount of spin-time. Furthermore, selecting a fixed-spin time poses difficulty: a spin-time of t is poor for parallel programs with a load-imbalance of $t+\epsilon$. For this reason, we investigate *adaptive blocking* algorithms that adjust the spin-time based on observed program behavior. We begin by establishing our performance target with an algorithm that has knowledge of a program's load-imbalance, and then present two algorithms that estimate load-imbalance at run-time.

6.1 Load-Imbalance Oracle

Our first step towards developing a realistic adaptive two-phase blocking algorithm is to consider the appropriate spin-time before reads and barriers given knowledge of the maximum load-imbalance across processes. Our oracle blocking algorithm has the following behavior:

- There exists a minimum spin-time before blocking, S, that ensures that that coordinated processes remain in coordination. Processes arriving at a read operation always spin S.
- There exists a load-imbalance, V, past which it is more beneficial to block at a barrier than to spin until the barrier completes.
  - If the load-imbalance in the program, v, is greater than V, then processes spin for only S at barriers.
  - If v is less than V, then processes should spin for v plus the network round-trip time. Spinning for longer than v at a barrier implies that the processes are not currently coordinated.

In Section 5.2 we saw that blocking at barriers introduces a scheduling skew of $2 \cdot cs$ across processes. Thus, the minimum spin-time, S, is $2 \cdot cs$.

Determining V requires comparing the local benefit of a processor switching to another job and the global penalty experienced by the system when cooperating processes are not coordinated. This cost analysis differs from determining the optimal spin-time for lock synchronization: waiting for a lock involves only a local cost and each processor can act greedily. Therefore, when acquiring a lock, blocking is beneficial when the expected wait-time exceeds the context-switch cost.

When a process arrives at a barrier operation, the gain that a processor achieves by switching to another process is the amount of useful work that the competing process completes before the barrier finishes (our assumption being that the processor switches back to original process at this time). Returning to Figure 7, this benefit is the expected waiting time minus the context-switch cost and the required minimum spin-time: $(\frac{v}{2} - cs - S) = (\frac{v}{2} - 3 \cdot cs)$.

[3] The preceding discussion makes a number of simplifications. First, network latency, L, is omitted from the required spin-time. Processes must spin-wait the additional time for the read-response and the barrier-completion to travel through the network. Second, we ignore the possibility that the root process of the barrier may need to be scheduled before broadcasting the barrier-completion message.

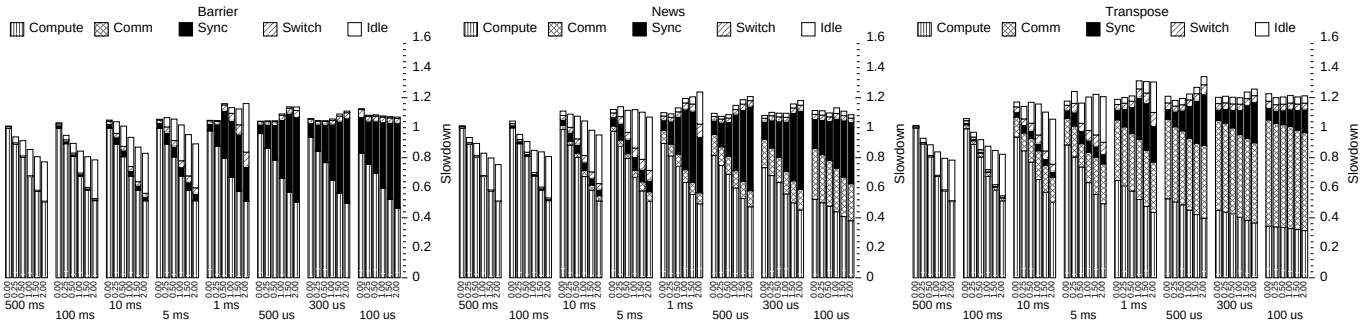

Figure 10: *Global Approximation of Program Load-Imbalance.* The slowdown of local scheduling with the global-adaptive algorithm relative to coscheduling is shown. The global algorithm calculates the load-imbalance accurately, using information available to the barriers within the parallel application. As a result, the performance is near that of the oracle algorithm.

The global loss to the system when a process blocks for a barrier is the resulting increase in the time for all processes to be scheduled after completing the barrier: $2 \cdot cs$. According to this simple analysis, blocking is beneficial to spinning when the global benefit exceeds the global cost: $(\frac{v}{2} - 3 \cdot cs > 2 \cdot cs)$ or $(v > 10 \cdot cs)$. This relation agrees with our simulation results for all three communication patterns, where we find that increasing the spin-time beyond 10 times the context-switch cost results in no improvement.[4]

The performance results for the algorithm described above, when the blocking algorithm knows the load-imbalance of the processes in the program, are shown in Figure 8. The performance over the entire range of data points is very near the best performance we see for each data point using its best fixed-weight spin-time. For the fine-grain versions of NEWS, the performance is within 10% of coscheduling for most data points and within 20% when the load-imbalance is between and 450 μs and 1500 μs. The worst-case performance occurs when the computation granularity is 1 ms and the load-imbalance is $2g = 2$ ms, which is when the load-imbalance is equal to $V = 10 \cdot cs$. For all load-imbalances greater than V our algorithm is equal to or superior to coscheduling.

6.2 Local Approximation of Load-Imbalance

In the following two sections we examine a local and a global algorithm for obtaining an approximation of an application's load-imbalance. In our first algorithm, each executing process approximates the load-imbalance in the program by sampling the time for its barriers to complete. If the scheduling of processes of the job are coordinated across processors, then each process will see that its average wait-time for the opening barrier is $\frac{v}{2} + 2 \cdot L$ and the time for the closing barrier is $2 \cdot L$. One method for each process to approximate v is this environment is to record the maximum waiting time it has seen up to this time.

However, in our implicitly scheduled environment, processes are not always coordinated; therefore, processes measure some waiting times that are much larger than v. Our heuristic is to record each of the wait-times and remove the largest 10% of the data points as outliers due to scheduling irregularities. After disregarding the top 10% of the data points, we use the largest remaining wait-time as our approximation of v. The algorithm adapts to changes in load-imbalance over the lifetime of a program and bounds the amount of necessary buffer-space by replacing wait-times selected at random from the current sample.

As shown in Figure 9, the local adaptive algorithm exhibits performance worse than the oracle algorithm for a number of fine granularity programs. The reason is obvious when we examine each process's approximation of the load-imbalance over time: the approximation is often less than the actual value; as a result, a process does not always spin long enough to wait through the barrier. The low approximation occurs because some of the valid wait times are eliminated from our sample as outliers. Rather than improving the approximation procedure employed by each independent process, we investigate a new method that uses global information available to the program.

6.3 Global Approximation of Load-Imbalance

Our next adaptive algorithm leverages the global communication occurring naturally within the program to estimate the program's load-imbalance. The key observation is that if the barrier is implemented in software, then the root process in a barrier can more accurately observe the distribution of waiting times than individual processes. In our barrier implementation, each process sends a message to a root process; when the root has received P such messages, it broadcasts a barrier-completion message to the waiting processes. The root process knows that the load-imbalance of a computation phase is equal to the longest waiting time of any given process. The root calculates the appropriate spin-time for barriers and propagates the result as part of the barrier-completion message. In this manner, each process is informed of the spin-time at a small additional cost. Once again, even from the perspective of the root process, scheduling irregularities are exhibited by inflated wait times; thus, when calculating the load-imbalance, the root ignores the top 10% of the maximum wait-times in the sample.

Figure 10 shows the performance of the global-adaptive blocking algorithm for the three communication patterns. As is evident from the middle graph (NEWS), global approximation achieves performance very similar to that of the oracle algorithm. As expected, all communication patterns with coarse-grained computation and high load-imbalance achieve better performance with globally adaptive blocking

[4] The adaptive two-phase algorithm that we described differs significantly from the algorithms, such as a random walk, described in [22] that worked well for lock synchronization. We found that these algorithms are not effective for barriers because they learn to immediately block, since spinning for any amount of time is not beneficial until the processes are coordinated. The algorithms would need to be modified to always spin for at least S.

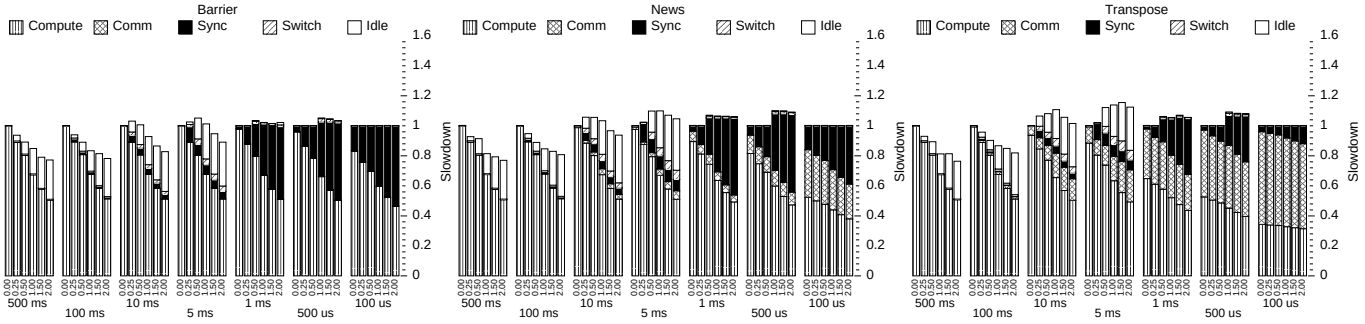

Figure 11: *Coordination of Timer Expiration.* The slowdown of the global-adaptive blocking algorithm relative to coscheduling improves by 20% when there is no skewing of timers across processors.

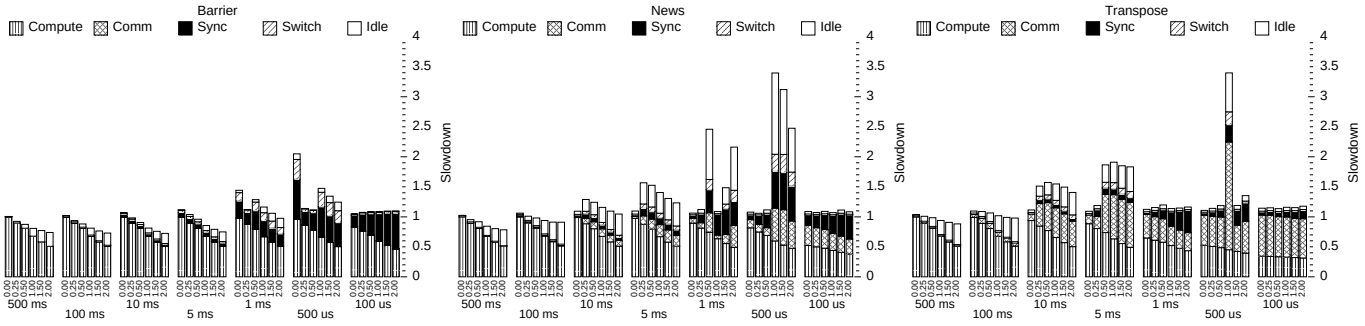

Figure 12: *Round-Robin Local Scheduler.* The relative slowdown of the global-adaptive blocking algorithm with a round-robin scheduler is significantly worse than with a SVR4 scheduler due to the lack of priorities and dynamic coordination.

and implicit scheduling than with coscheduling. At these points, the global algorithm learns to spin for only a short time and another process is scheduled. The largest slowdown occurs when the load-imbalance in the program is near V (*i.e.*, $v = 10 \cdot cs = 2\ ms = 2g$ and $g = 1\ ms$). Our final results indicate that we are able to achieve performance within 35% of coscheduling for heavily communicating and synchronizing programs.

7 Sensitivity to the Local Scheduler

In this section, we examine the effects of changes to the local scheduler on performance. First, we quantify the performance of global adaptive blocking when timers across processors are synchronized. Second, we demonstrate that using a round-robin local scheduler leads to unpredictable and unacceptable performance.

7.1 Timer Skew

As described in Section 3.3, there are two periodic timers in the SVR4 scheduler: a clock tick every 10 ms for time-quanta expiration and a timer each second to prevent the starvation of processes. The simulations up to this point have made the realistic, but pessimistic, assumption that these timers are independent across processors.

Figure 11 shows the performance of the global-adaptive blocking algorithm when timers expire simultaneously across processors. The worst-case performance is now only 15% slower than when coscheduled, compared to 30% when timer skew exists. The performance of the most fine-grain computations (*i.e.*, g=100 μs) is now identical to coscheduling.

There are many algorithms that perform fine-grained clock synchronization in a distributed system [3, 34]. Whether or not the implementation cost of synchronizing the machines is worth the performance benefits remains to be seen.

7.2 Round-Robin Scheduling

One particularly important component to local scheduling is the local scheduling algorithm running on each node in the system. Many previous studies have examined local scheduling with a round-robin local scheduler instead of a priority-based scheduler. We examine the impact of using a round-robin scheduler by simply altering the tunable tables of the SVR4 scheduler to have one quantum length and one priority.

The results in Figure 12 show that the round-robin scheduler is less robust than SVR4, with performance up to 3.4 times worse than coscheduling. This performance difference is due to absence of priorities in round-robin. As mentioned previously, in SVR4, processes returning from sleeping on an event have their priorities raised, and as a result are simultaneously scheduled across processors. With the round-robin scheduler this dynamic coordination after barriers only occurs when the newly unblocked process is the only runnable process on each processor.

8 Conclusions and Future Work

In this paper, we presented a simulation-based study of time-sharing approaches to parallel job scheduling. The two scheduling algorithms we compared were implicit scheduling and coscheduling. As opposed to the explicit methods of coscheduling for simultaneously scheduling processes

Barrier			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
Context-Switch	Latency		Low	High	Low	High	Low	High
Low	Low	High	1.09	1.08	1.00	0.83	0.92	0.81
		Low	1.01	1.06	1.01	0.84	0.93	0.77
High	High	High	1.12	1.05	1.07	0.94	0.93	0.81
		Low	1.08	1.07	1.07	0.95	0.94	0.81

News			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
Context-Switch	Latency		Low	High	Low	High	Low	High
Low	Low	High	1.26	1.21	1.03	0.91	0.92	0.80
		Low	1.02	1.23	1.03	0.93	0.93	0.79
High	High	High	1.12	1.10	1.11	1.09	0.94	0.81
		Low	1.12	1.09	1.13	1.10	0.93	0.81

Transpose			Computation Granularity					
			Fine		Medium		Coarse	
			Imbalance		Imbalance		Imbalance	
Context-Switch	Latency		Low	High	Low	High	Low	High
Low	Low	High	1.12	1.10	1.07	0.99	0.93	0.81
		Low	1.05	1.10	1.05	0.99	0.94	0.81
High	High	High	1.23	1.23	1.22	1.20	0.93	0.80
		Low	1.23	1.19	1.23	1.22	0.93	0.80

Figure 13: *Implicit Scheduling versus Coscheduling.* Each entry in the table shows the final slowdown of implicit scheduling with the global-adaptive blocking algorithm compared to coscheduling. The network latency is set to 10 μs and 100 μs for context-switch costs of 50 μs and 200 μs. Fine-, medium-, and coarse-grain programs are evaluated, where the computation granularity is 100 μs, 500 μs, and 500 ms rrespectively; the load-imbalance is set to 0.25 to 1.5 times the granularity. **Black** regions indicate where the performance of coscheduling is still 15% better than implicit scheduling; **gray** indicates implicit scheduling performs 15% better.

of a parallel job across processors, in an implicit system, each scheduler independently decides which process to run based on feedback from communication and synchronization events. Figure 13 shows the final comparison between implicit scheduling with the global adaptive two-phase blocking algorithm and an optimistic implementation of coscheduling for our three communication patterns. We see that, with no global information, implicit scheduling performs no worse than 25% slower than coscheduling for a range of computation granularities, load-imbalances, network latencies, and context-switch times. In fact, for coarse-grained programs with some load-imbalance, implicit scheduling is up to 25% better.

Previous research has shown that coscheduling is superior to local scheduling approaches for fine-grain programs. Our conclusions differ from these studies due to changes in the blocking algorithm and the local scheduler. We found that a simple two-phased algorithm performs reasonably well if the spin-time is at least equal to the skew introduced by the local schedulers (*i.e.*, twice the context-switch cost) or matches the load-imbalance of the parallel program. We also presented a new, global-adaptive blocking algorithm which uses barriers within the application to dynamically estimate the load-imbalance. This approach performs slightly better, and is more robust to changes in load-imbalance, than the fixed-spin algorithm. We found that priority-based, preemptive local schedulers dynamically coordinate communicating programs by giving a temporary boost to processes awakening from communication and synchronization events. This is an important feature lacking in simple round-robin schedulers.

Further work is needed to conclude that implicit scheduling is an equivalent or superior alternative to coscheduling. The simulation results presented in this paper have focused on a necessarily small subset of synthetic parallel applications. Studying the effect of small changes to our bulk-synchronous programming model (*e.g.*, considering programs with exponentially distributed load-imbalances or imbalances within the communication phase, and applications without an opening barrier) may prove to be enlightening, as would simulations on a more diverse workload (*e.g.*, message-passing programs and non-bulk-synchronous programs where competing jobs have different communication behaviors). Evaluating the impact of architectural features, such as a network interface capable of handling read requests, is also of interest.

A number of potential improvements to implicit scheduling are already emerging. We anticipate that applications whose load-imbalance changes over time will benefit from more sophisticated two-phase adaptive blocking algorithms. Programs that do not precede communication with a barrier operation may need an implicit scheduling approach that monitors message arrivals to coordinate communicating processes. Additional work is also required to ensure that competing programs with different computation granularities and load-imbalance are given a fair share of processing time; preliminary simulations indicate that fine-grain jobs relinquish the processor too frequently relative to more coarse-grain jobs, and, subsequently, are given less processor time.

Clearly, an implementation of implicit scheduling with measurements on real workloads is required. The workloads of interest include not only parallel applications, but also a mixture of interactive and I/O-intensive processes. Our hope is that an implementation of implicit scheduling on a network of workstations will be more portable, scalable, and fault-tolerant, and yet less complex, than a coscheduling implementation with equivalent performance.

Acknowledgments

The authors would like to thank Eric Anderson, Mike Dahlin, Seth Goldstein, Alan Mainwaring, and the anonymous reviewers for their many useful comments on improving the presentation of this material. We are also appreciative of the efforts of Doug Ghormley and Amin Vahdat, members of the Glunix team in the Network of Workstations project, toward implementing the environment that assisted in running many of our simulations.

References

[1] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[2] 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. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[3] K. Arvind. Probabalistic Clock Synchronization in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474–87, May 1994.

[4] M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and Algorithms for Co-scheduling Compute-Intensive Tasks on a Network of Workstations. *Journal of Parallel and Distrbuted Computing*, 16:319–327, 1992.

[5] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[6] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, Dec. 1995.

[7] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 34–44, Feb. 1994.

[8] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, Dec. 1991.

[9] D. Culler, L. T. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, Feb. 1996.

[10] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *20th Annual International Symposium on Computer Architecture*, May 1993.

[11] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, Penn., 1979.

[12] A. Dusseau, D. Culler, K. Schauser, and R. Martin. Fast Parallel Sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 1996. To Appear.

[13] S. Evans, K. Clarke, D. Singleton, and B. Smaalders. Optimizing Unix Resource Scheduling for User Interaction. In *1993 Summer Usenix*, pages 205–218. USENIX, June 1993.

[14] D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.

[15] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec. 1992.

[16] D. G. Feitelson and L. Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, Apr. 1995.

[17] D. Ghosal, G. Serazzi, and S. K. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[18] B. Goodheart and J. Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[19] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–131, May 1991.

[20] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993.

[21] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–18, Nov. 1993.

[22] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Thirteenth ACM Symposium on Operating Systems Principles*, Oct. 1991.

[23] K. Keeton, T. Anderson, and D. Patterson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Proceedings of Hot Interconnects III*, Aug. 1995.

[24] L. I. Kontothanassis and R. W. Wisniewski. Using Scheduler Information to Achieve Optimal Barrier Synchronization Performance. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[25] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1990.

[26] S. T. Leutenegger and X.-H. Sun. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Proceedings of Supercomputing 93*, pages 143–152, 1993.

[27] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 226–236, May 1990.

[28] R. P. Martin. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of Hot Interconnects II*, July 1994.

[29] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multi-programmed Shared-Memory Multiprocessors. In *ACM Transactions on Computer Systems*, volume 11, pages 146–178, May 1993.

[30] C. McCann and J. Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, Feb. 1994.

[31] C. McCann and J. Zahorjan. Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 208–219, May 1995.

[32] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Winter USENIX*, Jan. 1996.

[33] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings of Supercomputing '93*, pages 824–833, Nov. 1993.

[34] A. Olson and K. G. Shin. Fault-tolerant Clock Synchronization in Large Multicomputer Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):912–923, Sept. 1994.

[35] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[36] V. G. Peris, M. S. Squillante, and V. K. Naik. Analysis of the Impact of Memory in Distributed Parallel Processing Sytems. In *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, Feb. 1994.

[37] K. C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[38] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

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

[40] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 40–51, Dec. 1995.

[41] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[42] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 199–206, July 1995.

[43] J. Zahorjan and E. D. Lazowska. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proceedings of the IFIP International Seminar on Performance of Distributed and Parallel Systems*, pages 455–472, Dec. 1988.