To appear inn *Proceedings of 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

A Closer Look At Coscheduling Approaches for a Network of Workstations*

Shailabh Nagar *Ajit Banerjee* *Anand Sivasubramaniam* *Chita R. Das*

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802.
Phone: (814) 865-1406
anand@cse.psu.edu

Abstract

Efficient scheduling of processes on processors of a Network of Workstations (NOW) is essential for good system performance. However, the design of such schedulers is challenging because of the complex interaction between several system and workload parameters. Coscheduling, though desirable, is impractical for such a loosely coupled environment. Two operations, waiting for a message and arrival of a message, can be used to take remedial actions that can guide the behavior of the system towards coscheduling using local information. We present a taxonomy of three possibilities for each of these two operations, leading to a design space of 3×3 scheduling mechanisms. This paper presents an extensive implementation and evaluation exercise in studying these mechanisms.

Adhering to the philosophy that scheduling and communication are intertwined and should be studied in conjunction, a complete communication substrate for UltraSPARC workstations, connected by Myrinet and running Solaris 2.5.1, has been developed. This platform provides the entire Message Passing Interface (MPI) to readily run off-the-shelf MPI applications by employing protected low-latency user-level messaging. Several applications can concurrently use this interface. This platform has been used to design, implement, and uniformly evaluate nine scheduling strategies with a mixture of concurrent real applications with varying communication intensities. This includes four new schemes (Periodic Boost, Periodic Boost with Spin Block, Spin Yield, Periodic Boost with Spin Yield) that are presented in this paper. In addition to evaluating the pros and cons of each mechanism in terms of throughput, response time, CPU utilization and fairness, it is shown that Periodic Boost is a promising approach for scheduling processes on a NOW.

*This research is supported in part by an NSF Career Award MIPS-9701475, NSF grant MIP-9634197 and an NSF equipment grant CDA-9617315.

1 Introduction

Networks of Workstations (NOW) have emerged as a cost-effective solution to high performance computing. As with any other parallel machine, two important issues limit their delivered performance. First is the hardware and software cost of communicating between processes executing on different nodes of the system. Second is the wastage of CPU time due to non-ideal scheduling of these communicating processes.

The problem of lowering communication overhead for a NOW has drawn a lot of attention. On the hardware side, high bandwidth networks such as Myrinet [6] and ATM promise to handle the high data rates of demanding applications, with point-to-point latencies comparable to those provided by interconnection networks of custom-built parallel machines. On the software side, low-latency user-level messaging substrates (such as U-Net [23] and Fast Messages [17]) have been developed using the intelligent network interfaces provided by these networks. Approaches to translate these improvements in message latencies to the applications in the form of efficient application-level messaging layers such as MPI [14] have been undertaken [25, 12].

Optimizing communication alone may not necessarily translate to improved performance since the scheduling strategy could nullify any savings. For instance, a currently scheduled process on one node would experience a long wait for a message from a process not currently scheduled on another node regardless of the low latency for messages. Scheduling and communication are thus closely intertwined, and should be studied together. This paper adheres to this philosophy when implementing an execution platform for real applications on a NOW.

Scheduling of processes onto processors of a parallel machine has always been an important and challenging area of research. Its importance stems from the impact of the scheduling discipline on the throughput and response times of a system. The research is

challenging because of the numerous factors involved in the design and implementation of a scheduler. Some of these influencing factors are the parallel workload, presence of any sequential and/or interactive jobs, native operating system, node hardware, network interface, and communication software. Previous studies on parallel schedulers have focussed their attention on closely coupled parallel systems. Communication and synchronization costs on such machines are relatively low, making complex schemes feasible. However, many of these scheduling schemes are not very practical for a loosely-coupled NOW environment.

Scheduling is usually done in two steps. The first step is assigning a process to a processor, and the second is scheduling the processes assigned to a processor. There is a considerable body of literature [19, 13, 18, 26, 22] related to the first step on closely coupled multiprocessor systems. Some of these studies [22, 26, 13] exploit the relatively low communication and synchronization overheads of these machines, particularly those with shared memory capabilities, to dynamically move processes across processors based on CPU utilization. Other studies , however, assume process migration to be expensive and propose static processor allocation strategies in which the set of processors is spatially partitioned. On a network of workstations environment, communication and synchronization costs are relatively high. Further, processes, as implemented by the native operating system at each workstation, are heavyweight, making it expensive to migrate them. Hence, process migration is cost-effective only for relatively long running jobs [1].

The second scheduling step, which is perhaps more important for a NOW environment, is the scheduling of assigned processes at each workstation. The choices here range from scheduling strategies based purely on local knowledge at a workstation to those using global knowledge across workstations. Local scheduling, which does not require any global knowledge, is relatively simple to implement. In fact, one could leave the processes to be scheduled by the native operating system of the workstation. The drawback is that the lack of global knowledge can result in lower CPU utilization and higher communication or context switching overheads. At the other end of the spectrum is *coscheduling* (also called gang scheduling) [16, 9, 10], which schedules processes of a job simultaneously across all processors, giving each job the impression that it is running on a dedicated system. While coscheduling has been shown to be essential for the efficient performance of fine-grained parallel applications, an implementation on a NOW can become expensive. Further, from performance and reliability perspectives, it does not scale well with the number of nodes in the system.

Another class of scheduling strategies use communication behavior (which are local events at each node) to guide the scheduling. Studies like [21, 2, 8, 4] attempt to dynamically coschedule communicating processes to improve job performance. Only two of these dynamic approaches [4, 20] have been proposed, implemented and evaluated on an actual NOW environment. These two strategies, called implicit coscheduling [4] and dynamic coscheduling (DCS) [20, 7], use information available locally to estimate what is scheduled on the other nodes without requiring any explicit messages for obtaining this information. Two actions, namely, *waiting for a message* and *receipt of a message*, form the basis for these schemes. Implicit coscheduling is based on the heuristic that a process waiting for a message should receive it in a reasonable time (as determined by the message latency and other factors) if the sender is also scheduled currently. Dynamic coscheduling, on the other hand, uses message arrival to indicate that a process of the same job (the sender) is scheduled on a remote node and schedules the receiver accordingly. The former [4] has been implemented and evaluated on Active Messages [24], which offers a closer coupling between the sender and receiver processes than MPI on Fast Messages [17], which has been used in evaluating DCS [20]. Further, the version of Fast Messages used in [20] can handle only one parallel application per node, and as a result, the evaluation is rather limited.

These studies raise some important questions for NOW schedulers. First, what is the design spectrum for developing communication based dynamic scheduling mechanisms on a NOW? In particular,what are the pros and cons of scheduling using message wait and message arrival information ? Second, how can these scheduling techniques be implemented within the context of current user-level messaging platforms (where the OS scheduler is unaware of communication events) ? Third, how do these schemes compare with each other and with the ideal behavior in terms of throughput, response time and fairness ? Answers to these questions require an experimental testbed to design and implement various scheduling strategies and a detailed evaluation to understand the intricate interaction between several factors. To our knowledge, no previous study has extensively evaluated these issues on a unified framework.

In this paper, we attempt to answer some of these questions. We present a unified taxonomy for classifying different approaches to waiting for a message and handling message arrival, leading to a design space of nine scheduling mechanisms. This includes five new mechanisms, called *Periodic Boost* (PB), *Periodic Boost with Spin Block* (PB-SB), *Spin Yield* (SY), *Periodic Boost with Spin Yield* (PB-SY), and *Dynamic Coscheduling with Spin Yield* (DCS-SY), in addition to the already existing schemes, namely Spin Block (SB), Dynamic Coscheduling (DCS), and Dynamic Coscheduling with Spin Block (DCS-SB).

We implement and evaluate the nine scheduling mechanisms on a testbed of SUN UltraSPARC server machines running Solaris 2.5.1, connected by Myrinet [6]. Using a protected, user-level communication substrate (U-Net), we have implemented the entire MPI messaging layer so that several off-the-shelf applications can be readily used for evaluations. The implementation exercise has involved writing software for the Myrinet interface card, user-level libraries, and kernel drivers, without requiring any modifications to the Solaris kernel. We conduct an exhaustive evaluation of the nine scheduling mechanisms with a mixture of multiple MPI applications, having varying communication granularities, executing at each node.

The results show that for workloads with low communication intensities, there is little difference between the scheduling schemes. As communication increases, there is clearly a need for a scheme which uses some heuristic to guide the system towards coscheduling. Of the schemes considered,it is observed that PB outperforms most other mechanisms over a range of different workloads in terms of the overall system throughput (total completion time divided by the number of jobs serviced), and response time. PB is also reasonably fair when we consider workloads with similar communication intensities. However, PB can unfairly favor higher communication jobs when we consider mixed workloads. This is analogous to the traditional multi-level priority-based UNIX System V scheduler, which can unfairly favor I/O bound jobs in a mixture of CPU and

I/O bound jobs. DCS-SB, DCS-SY and DCS are also good candidates to provide improved performance. In addition, we show that SY can be used as an alternative to SB in augmenting certain scheduling strategies.

The rest of this paper is organized as follows. Section 2 classifies the scheduling schemes and briefly describes their implementation. A description of the evaluation methodology, performance results comparing the scheduling disciplines, and implication of these results is presented in Section 3. Finally, Section 4 concludes with a summary of results and identifies directions for future research.

2 Scheduling Strategies

What do you do on message arrival?	How do you wait for a message?		
	Busy Wait	Spin Block	Spin Yield
No Explicit Reschedule	Local	SB	SY
Interrupt & Reschedule	DCS	DCS-SB	DCS-SY
Periodically Reschedule	PB,PBT	PB-SB	PB-SY

Table 1: Design space of scheduling strategies

Logically, there are two components to the interaction between the scheduler and the communication mechanism. The first is related to how a process waits for a message. This can involve: (a) just spinning (busy wait); (b) blocking after spinning for a while; or (c) yielding to some other process after spinning for a while. The second component is related to what happens when a message arrives and is transferred to application-level buffers. Here again, there are three possibilities: (a) do no explicit rescheduling; (b) interrupt the host and take remedial steps to explicitly schedule the receiver process; or (c) periodically examine message queues and take steps as in (b). These two components can be combined to give a 3×3 design space of scheduling strategies shown in Table 1.

To implement the strategies described, we have written or modified three major software components. The first is the LANai control program (LCP) executing on the LANai processor of the Myrinet interface. This program performs the data transfer between the host memory and the network. Though this can raise an interrupt for the host processor, this feature is not used for regular message transfer which uses polling at the user level. The second is a set of user-level libraries for a messaging layer that provides the complete MPI [14] functionality which is based on the MPICH distribution [11]. Details of its implementation and performance are not included here due to space limitations and the reader is referred to [5]. The set of libraries includes U-Net [23] from Cornell, together with umlib and MPI Unet which we have developed to provide the efficient MPI interface. The implementation incorporates several optimizations to eliminate multiple levels of copying. The routines here manage the send and receive queues mapped directly into user address space to avoid any kernel invocations for data transfers. The third component is a kernel device driver, which in the baseline implementation is used only at the initialization stage to set up endpoints. An endpoint is a virtual network interface that provides a process a handle into the communication mechanism. The device driver also offers the potential for performing some actions in kernel mode (via an ioctl call), used in implementing certain scheduling schemes. A schematic showing the different components in the baseline implementation (Local) and potential additions for implementing the different scheduling strategies is shown in Figure 1.

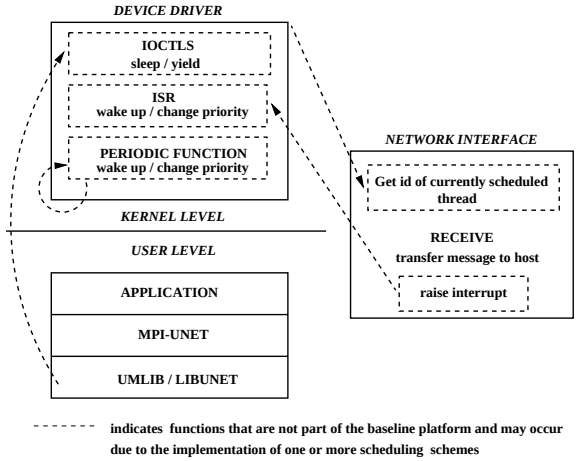

Figure 1: Software components used in the implementations

In the following discussion, we present a brief description of each scheduling strategy considered in this study and its implementation on our platform. A more detailed description can be found in [15].

1. **Local** : This scheme has been considered as a baseline to show the need for a scheduling strategy based on global information. The scheduling of the parallel processes on each node is solely determined by the native Solaris scheduler and is no different from that of a serial process running on the same node. A process spins on a message receive and incoming messages are directly transferred to user-level buffers by the network interface. From the implementation viewpoint, this scheme is straightforward since no modifications/additions need to be performed.

2. **Spin Block (SB)** : Here a process spins on a message receive for a fixed amount of time before blocking itself. The fixed time for which it spins, henceforth referred to as *spin time*, is carefully chosen to optimize performance. The rationale here is that if the message arrives within the spin time, the sender process can be assumed to be scheduled on its node and hence the receiver should hold on to the CPU so as to be coscheduled with the sender. Otherwise, it should block so that CPU cycles are not wasted. While a theoretical analysis to calculate the optimal spin time can be done in a few situations (as in [4]), such an analysis can become exceedingly complex for a real application running on a generic message passing layer such as MPI. We have resorted to an empirical approach to quantify the optimal spin time for a given application (similar to [20]). The SB mechanism is different from a similar mechanism implemented with implicit coscheduling in [4]. The latter is implemented on Split-C/Active Messages [24], which inherently has a tighter coupling (than MPI) between a sender and a receiver process. Moreover, it uses an additional $pairwise$ spin time to synchronize each pair of communicating processes.

Spin Block improves performance in two ways. First, by reducing the number of CPU cycles spent in idle spinning, it increases CPU utilization per node. Second, due to the priority boost a process can receive on wakeup, it is more likely to be scheduled soon after getting a message. Since the sender of the message is also likely to be scheduled at that time (one way latencies being much smaller than an average time slice), the probability for a pair of communicating processes to run simultaneously for some time increases.

In our implementation of Spin Block, the polling loop in the user-level library receive call is modified to run until *spin time* elapses. If a message still does not arrive, an ioctl call is made to the kernel device driver, which makes the process block on a semaphore. The ioctl routine also registers a wakeup call for the corresponding endpoint with the network interface. When a message arrives for that endpoint, the LANai issues an interrupt, and the interrupt service routine (ISR) signals the corresponding semaphore. The Solaris signaling mechanism moves the blocked process to the head of the runnable queue for a higher priority level most of the time. The woken up process can thus get a priority boost on receipt of a message (though this is done implicitly within Solaris and not explicitly by our code).

3. **Dynamic Coscheduling (DCS)** : DCS also uses incoming messages as an indication that the sender is scheduled on its node. Unlike SB, it uses only a busy-wait for receiving. However, if the receiver is not running, it *explicitly* schedules it by hiking its priority.

   Our implementation of DCS is similar to the one discussed in [20]. The library level of the messaging platform is left unchanged. The LANai periodically gets the id of the thread currently executing on the host CPU. On receipt of a message, the LANai checks whether the message destination process matches that currently estimated to be running. If there is a mismatch, an interrupt is raised. The interrupt service routine, after verifying again that the mismatch exists, boosts the priority of the destination process causing it to be scheduled almost immediately.

   DCS does not handle application level skews (due to work imbalance) between the sender and receiver, and lets processes spin for the remainder of their time slices in such situations. However, since it boosts priorities of receiver processes when they are spinning or even earlier (if not scheduled), it can potentially coschedule communicating processes more often (and sooner after message arrival) than Spin Block which implicitly boosts priorities only on wakeup (and not during spinning).

4. **Dynamic Coscheduling with Spin Block (DCS-SB)** :

   In DCS there is a potential for CPU cycles to be wasted in spinning for a message which does not come soon. It can be combined with Spin Block to limit this wastage to the spin time. DCS-SB does exactly that. Spinning receivers go to sleep after a fixed spin time has elapsed. Incoming messages cause destination processes to either have their priority explicitly boosted (if they are not blocked) or be woken up.

   DCS-SB incurs the overheads of both Spin Block and DCS, though not always together. Its ability to increase coscheduling beyond what DCS can offer is more significant at a higher load (when there is other useful work to utilize the saved CPU cycles) at a workstation.

   User-level library changes for DCS-SB are identical to those for SB. The LCP raises an interrupt not only for mismatches between estimated scheduled process and destination process but also when the destination is known to have blocked. The ISR, correspondingly, checks both these conditions and does a wakeup or priority boost as needed.

5. **Periodic Boost (PB) and Periodic Boost with Timestamps (PBT)** :

   The first of the newer schemes that we propose is called Periodic Boost (PB). Going back to either of the DCS schemes or SB, we observe that the solution to approach gang scheduling has been to boost the priority of the process (the destination of a message) on message arrival. However, this boost is done within the interrupt service routine since an interrupt is the only way of detection of message arrival by the kernel in these schemes. When there is a mixture of high communication workloads running at a workstation, these schemes can result in a large number of interrupts, negating the purpose of user-level messaging. In the PB scheme, we propose that we do not have any interrupts being raised at all. Rather, we can have an entity (thread) within the kernel which periodically examines the endpoints of the parallel processes and boosts their priorities. Though a number of criteria can be used for boosting priorities, we use two, which leads us to the two schemes called PB and PBT. In PB, the periodic mechanism checks the endpoints in a round-robin fashion and boosts the first with an unconsumed (henceforth called pending) message; if no one has pending messages, no one is boosted. In PBT (Periodic Boost using Timestamps), the periodic mechanism boosts the process which has the most recently arrived pending message. The frequency with which these actions should be taken needs to be chosen carefully. We have used an experimental approach to find this frequency and have found that invoking the boosting function once every 10 milliseconds gives good performance.

   PB and PBT are simpler to implement than the previous three schemes. There is absolutely no change in the user-level messaging libraries or in the LANai Control Program from the baseline implementation. All that is needed is an additional function in the device driver which gets called periodically (via a timer mechanism) to examine the number and/or the timestamp of pending incoming messages for each endpoint and boosts the priority of a process when needed.

   The potential benefits of PB are threefold. First, it allows a more complex heuristic (not necessarily based on communication information alone) to be used for making scheduling decisions for parallel processes. Second, since there is no additional work done by the LANai, the overhead for normal send/receive operations is minimized. Third, it is possible to dynamically control the invocation of the function (in the device driver), allowing the granularity of scheduling and the overhead to be controlled better. However, we do not fully

explore the first and third benefits in this paper, and base the priority boost purely on pending message information.

6. **Periodic Boost with Spin Block (PB-SB)** :

   This is an extension to PB (or PBT) motivated by the same reasons as were given in the DCS-SB case. Here, the time wasted in spinning is limited to the period of the kernel function which boosts priorities.

   From the implementation viewpoint, the user-level libraries, LCP and the ISR are identical to those used for Spin Block. There is a slight change in the kernel function which implements the PB mechanism. It now preferentially wakes up sleeping (blocked) processes which have pending messages. If there are none, it does priority boosting as in the PB case.

   PB-SB incurs the overheads of both PB and SB. So it can be expected to do better than PB when the load is high and there is other useful work to be done.

7. **Spin Yield (SY)** :

   In SB, the blocking of processes after spinning has two consequences. First, an interrupt is required to wake the process on message arrival (which is an overhead). Second, the block action only relinquishes the CPU and there is no hint given to the underlying Solaris scheduler as to what should be scheduled next.We attempt to fix these two problems using the Spin Yield (SY) scheduling strategy, Here, after spinning for the required spin time, the receiver process lowers its own priority, boosts that of some other process (based on pending messages) and continues to spin. This avoids an interrupt (since the process keeps spinning albeit at a lower priority), and gives more control over which process is scheduled next.

   The SY scheme only requires an ioctl call (compared to the baseline) after the spin time has elapsed. The ioctl needs to change the priority of the caller instead of putting it to sleep (as is done in SB).

8. **Dynamic Coscheduling with Spin Yield (DCS-SY)** :

   SY is an alternative to SB. Just as we had DCS-SB and PB-SB combinations, we could also have DCS-SY and PB-SY combinations.

   The implementation of DCS-SY takes the implementation of DCS and adds the functionality of SY. After the process performing a receive operation spins for a fixed interval, it yields i.e. lowers its priority and raises the priority of another process which has pending messages. The operations for the LANai and the driver remain the same as in DCS.

9. **Periodic Boost with Spin Yield (PB-SY)** :

   The implementation of PB-SY takes the implementation of SY and adds the extra function (which gets called periodically) in the driver to implement PB.

   From the performance viewpoint, the relative benefits of PB-SB and PB-SY directly translate from the relative benefits of SB and SY identified earlier.

The implementation of the novel scheduling strategies presented here can, however, have a detrimental effect on the normal data transfer mechanism. To investigate if there is indeed a slowdown in message latencies, we run a one-way latency benchmark with each scheduling scheme in place. Contrary to expectations, we find [15] that the one-way MPI latency (32 μs for 4 byte messages) is not significantly impacted by the implementation of any of these scheduling strategies.

3 Performance Results

We conduct a comparison of the schemes presented in the previous section on a uniform platform, using workloads that are a mixture of jobs with varying communication intensities. Our workloads are mixtures of parallel jobs and do not consider any explicit sequential/interactive ones. However, there are always some background/daemon processes executing on a workstation (even on an unloaded system) which can potentially perturb the execution of the parallel jobs.

3.1 Experimental Setup and Workloads

Our experimental platform is a network of eight Sun Ultra-1 Enterprise servers running an unmodified Solaris 2.5.1 operating system. The workstations have 167 MHz UltraSPARC processors with 64 MB of main memory and a 32 bit SBUS interface operating at 25 MHz. The eight workstations are connected by Myrinet through an 8-port switch with the interface cards having a 37.5 MHz LANai processor and 1 MB of SRAM.

The first application that we consider is LIFE, an example program that comes with the MPICH distribution, which is illustrative of near-neighbor communication in matrix computations. It simulates the game of life on a two-dimensional matrix of cells which is partitioned amongst the processors. Each processor communicates with its four nearest neighbors along the boundary of the sub-matrix assigned to it. More importantly, from the scheduling perspective, the application is of the bulk synchronous type with distinct communication and computation phases and a barrier separating the iterations. LIFE is particularly suitable for our study because by varying two parameters, namely problem size (matrix size) and the number of iterations, it is possible to control the granularity of communication while keeping the total execution time roughly the same. A large matrix size with small iterations results in a coarse grain application while a small matrix size with more iterations has fine grain communication characteristics.

Three other applications that we consider (MG, LU and EP) are from the NAS benchmark suite. MG is a simple multi-grid solver that solves constant coefficient differential equations on a cubical grid. It is the most communication intensive of the three and spends 26% of the execution time on communication. LU is a matrix decomposition application that uses a large number of small messages. Of the three, it falls in the middle in terms of communication intensity with 16% of the execution time on communication. EP is an embarrassingly parallel application that is typical of many Monte-Carlo simulations. There is very little communication in this application ($<$1%) in the form of some global sums towards the end of a large computation. For the purposes of this study, it only serves as a competitor for processor cycles which can skew the scheduling of other communicating parallel applications.

Using these four applications, we first construct nine different workloads (shown in Table 2), each containing four applica-

Wkld	Applns. in Workload	Comm.
1	LIFE (3.5), LIFE (3.5), LIFE (3.5), LIFE (3.5)	(lo,lo,lo,lo)
2	LIFE (3.5), LIFE (3.5), LIFE (3.5), LIFE (12)	(lo,lo,lo,hi)
3	LIFE (3.5), LIFE (3.5), LIFE (12), LIFE (12)	(lo,lo,hi,hi)
4	LIFE (3.5), LIFE (12), LIFE (12), LIFE (12)	(lo,hi,hi,hi)
5	LIFE (12), LIFE (12), LIFE (12), LIFE (12)	(hi,hi,hi,hi)
6	EP (<1), EP (<1), EP (<1), EP (<1)	(lo,lo,lo,lo)
7	MG (26), LU (16), LIFE (12), EP (<1)	(hi,me,me,lo)
8	MG (26), MG (26), EP (<1), EP (<1)	(hi,hi,lo,lo)
9	MG (26), MG (26), MG (26), MG (26)	(hi,hi,hi,hi)

Table 2: Four Process Mixed Workloads (% of time spent in communication is given next to each application)

tions, which capture interesting mixes of the applications. The middle column shows the four chosen applications for the workload and the percentage of communication (of the total execution time) in that application. The third column gives a quick overview of the mix of communication intensities of the applications in the workload (*lo* indicates relatively low communication, *hi* indicates relatively high communication, and *me* is in between). The first five workloads are constructed directly from the LIFE application which provides tunable parameters to vary the communication intensity. They range from all four processes at a workstation having low communication, through a mix of high and low communication intensities, to a fully communication heavy workload. The next four workloads choose a mix of the four applications, and again span from low to high communication intensities.

The problem size for the different applications are adjusted so that each takes approximately the same time (25 seconds) to complete if it were run alone, and they are reasonably small so that all of them can simultaneously fit in primary memory (to minimize paging effects). For instance, executing the four job workloads on an ideal gang scheduled environment (without any overheads for scheduling) would result in a total completion time of 100 (4 * 25) seconds for workloads 1 through 9. Executing them together, however, increases the completion time of each instance by an amount that is dependent on the chosen mix and the scheduling scheme.

There are several criteria – such as throughput/utilization, average response/turn-around time, variance in response times, fairness, and degree of coscheduling – that can be used to qualify or quantify the performance of a scheduling scheme. While one could argue that the degree of coscheduling should be used to compare the scheduling schemes outlined here (because they try to approximate the behavior of coscheduling), the bottom line from the system designer's perspective is to maximize the throughput/utilization of the system while maintaining fairness (an equal/fair allocation of the CPUs to the jobs during execution). Similarly, the user is interested in minimizing the average response/turn-around time and its variance. Coscheduling is one way of meeting the system designer and user goals, but is not necessarily the only solution. In this paper, we examine performance from the perspective of the system designer and user.

In the first set of results, the metric we use is the time taken for the last process to complete since the first process started executing (which we call the *completion time*). The lower this time the more effective the scheduling scheme. Of course, an ideal coscheduling implementation would give the lowest completion time in most cases. Any additional time taken beyond this lowest completion time is considered a overhead. Therefore, if one of the above 4 process workloads were to take 140 seconds to complete on some scheduling strategy, it is said to have a slowdown of 40% over coscheduling. Hence, slowdown for a scheme (compared to ideal coscheduling) can be directly computed from the completion times given here. This time is also directly related to the system throughput (completion time divided by the number of jobs). In the next set of results, we give the completion times of the individual jobs to show the *variance in the turn-around times.* We also present figures monitoring the CPU utilization by each process during the course of an execution to discuss *fairness* issues. The reader should note that even though we have obtained detailed statistics to show some of the observations/claims stated below, we are not presenting them here due to space limitations.

3.2 Comparison of Scheduling Schemes

Table 3 shows the performance of the first five workloads using different scheduling strategies. Considering the schemes individually, the slowdown for Local even with Workload 1 is 80% (compared to coscheduling). The slowdown increases steeply as the workload becomes more communication intensive because of the well known problem of Local (lack of global knowledge in making scheduling decisions). Local's performance is not significant other than as a baseline to show the need for a more sophisticated scheduling policy that bases its decisions on what may be scheduled at other nodes.

	Workload				
	1	2	3	4	5
LOCAL	180	208	674	2524	3997
SB	124	153	773	1814	2849
DCS	162	192	350	533	764
DCS-SB	133	173	321	463	700
PB	130	138	152	226	451
PBT	152	190	252	295	284
PB-SB	130	158	655	1685	2660
SY	157	185	985	2320	3046
DCS-SY	166	205	347	527	717
PB-SY	141	158	287	459	733

Table 3: Completion Time in Seconds (Workloads 1 to 5)

SB, DCS and DCS-SB show a less steep increase in slowdown (compared to Local) as communication intensity increases. Between these three, we find that SB does better for workloads with lower communication but worse than the other two at higher communication intensities (workloads 3, 4 and 5). One possible reason for its poor performance for workloads 3 and above is the following. In SB, blocked processes get woken up (via the interrupt service routine) on arrival of a message. Due to the policies of the default Solaris scheduler, these processes mostly receive a priority boost on being woken up. Since DCS boosts the priority of the destination process of a message even if it has not yet called the receive function or when it is spinning but switched out (and not just when it blocks as is done in SB), any reply from the destination in DCS is likely to be sent back faster (thus increasing the likelihood of being coscheduled). DCS-SB, which combines the benefits of DCS (immediate priority boost of the destination process) and SB (limited

cycles wasted in spinning), performs even better than DCS.

It should be noted that the SB mechanism is different from implicit coscheduling presented in [4]. SB does not implement a pairwise spin component as in [4]. It also runs using a different programming model and messaging layer (MPI and UNet respectively) than [4], which uses Split-C over Active Messages. In Active Messages, the equivalent of a send causes a reply to be sent back by a handler at the remote node. So a receive equivalent following the send can have a better estimate of what is scheduled at the remote node. A corresponding send followed by receive in MPI cannot distinguish between load imbalance and scheduling skews. These factors make it difficult to directly compare the performance of SB presented here with the results for implicit coscheduling presented in [4].

Contrary to expectations, we find SY not performing as well as SB. There are two possible reasons for this. Spinning, despite lowering of priority, instead of blocking may eat away valuable CPU resources. More significant than this is the fact that the priority boost for a destination process of a message is done only when some other process does a receive at that node (and its spin time has expired). This may delay the priority boosting action even further than when it would have happened in spin block, thereby further delaying the reply message. This effect may outweigh the potential benefits of avoiding interrupt processing costs. This suggests that SY should not be used in isolation, but only in conjunction with some other mechanism which boosts the priority much sooner after message arrival. PB-SY and DCS-SY are two such solution approaches. In fact, PB-SY performs better than many schemes for several configurations. DCS-SY performs quite similar to DCS with a small improvement shown for higher communication intensities (when the savings of yield over block are more apparent).

Uniformly, we find that the Periodic Boost (PB) scheme, proposed in this paper, outperforms almost all other schemes and across all workloads. Even the rate of increase of slowdown (from 30% for workload 1 to 126% for workload 4) is much lower than the rate of increase for the other schemes. As a result, while it does better than the others for a low communication intensity mix, it does even better (compared to the other schemes) at higher communication intensities. Adding SB to PB does not seem to help significantly, while adding the overheads of blocking and interrupt processing costs. This suggests that we should not use SB in conjunction with PB.

PBT performs worse than PB for all but the highest communication workload. This is is most likely due to the overheads in the scheme. We have also observed that the PBT mechanism is extremely sensitive. Since it uses time-dependent information in making scheduling decisions, its results tend to vary significantly from one run to another. Hence, one should be cautious in making strong pronouncements about the performance of PBT.

Moving to Table 4, which uses mixtures of different applications, we can see many of the same patterns that were observed in Table 3. For workload 6 (4 instances of EP), the communication is so low that there is negligible difference between the scheduling schemes. Even Local does as good as any smart scheduling strategy, and there is a slowdown of only around 6% over coscheduling. Even though at the beginning of this section we mentioned that we are not explicitly running sequential jobs concurrently with parallel jobs, this result suggests that EP can be considered a sequential job for most practical purposes. The workloads with EP can thus be viewed as a mix of sequential and parallel jobs.

	Workload			
	6	7	8	9
LOCAL	106	1115	1088	3393
SB	104	438	301	1344
DCS	104	214	144	664
DCS-SB	101	221	134	525
PB	103	174	176	355
PBT	112	125	136	211
PB-SB	106	411	331	1467
SY	107	936	818	2674
DCS-SY	105	183	145	618
PB-SY	103	171	145	836

Table 4: Completion Time in Seconds (Workloads 6 to 9)

In Table 4, we again find that PB and PBT outperform all other scheduling strategies in terms of the slowdown over coscheduling and controlling the rate of increase of slowdown with increased communication. DCS-SB and DCS come next, with PB-SY close behind. Once again, SB does not do as well at higher communication workloads.

As mentioned earlier, the completion time of the last job of a workload may not necessarily be the only metric of importance. While this is important when looking at the throughput of the system, the user (and even the system designer) is interested in lowering the average and variance of turn-around times together with ensuring that a fair share of the CPU is allocated to each process during execution. We should thus examine the completion times of each of the jobs in a workload and their variance, as well as closely observe how the CPU(s) are allocated to the different jobs during the course of an execution.

	Workload 3					
	hi	hi	lo	lo	Mean	Coeff. of Var.
SB	773	760	139	137	452	0.80
DCS	350	344	188	186	267	0.35
PB	150	152	54	137	123	0.38
PBT	252	198	203	210	215	0.11
PB-SY	256	269	287	286	274	0.05
	Workload 5					
	hi	hi	hi	hi	Mean	Coeff. of Var.
SB	2845	2836	2849	2807	2834	0.01
DCS	764	764	755	739	755	0.02
PB	287	451	449	409	399	0.19
PBT	63	284	264	49	165	0.76
PB-SY	719	721	733	719	723	0.01

Table 5: Individual Completion Times for Workloads 3 and 5 (in secs)

To examine these issues, we focus specifically on workloads 3 and 5, and the performance of SB, DCS, PB, PBT and PB-SY (due to space limitations). Workload 3, with two lo and two hi jobs, and Workload 5, with all four hi jobs, would bring out the effect of heterogeneous and homogeneous communication intensity jobs on the different schemes. Table 5 shows the completion times for the individual jobs in the two workloads, the mean completion time

and the coefficient of variation (standard deviation divided by the mean). In addition, Figures 2 and 3 show the percentage allocation of the CPU to the different jobs at different points in the execution at a representative workstation. This has been found by periodically probing for the CPU time allocated to each job and dividing by the probe interval. It should be noted that the completion times in Figures 2 and 3, and Table 5 may not match and may be different from the ones presented earlier because they have been collected at only one representative workstation (and not necessarily at the machine where the maximum time is incurred). The applications do not begin execution at the origin of the X-axis in the graphs. From the user's perspective, a low mean completion time and a low coefficient of variation in completion times is desirable. From the fairness point of view, one would like to see equal CPU allocations to the current jobs in the system within each probe interval.

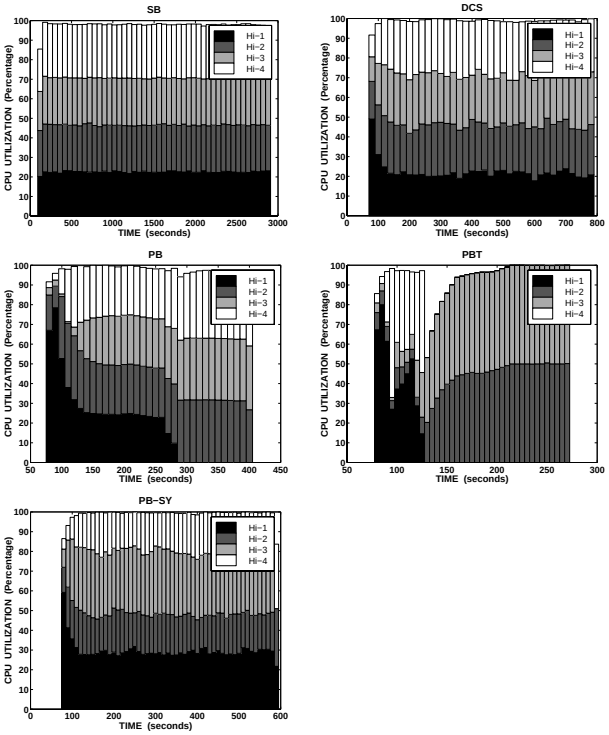

Figure 2: Monitoring CPU Utilization for Workload 5

Focusing first on the homogeneous workload (Workload 5), the four schemes other than PBT have a relatively low coefficient of variation. Of these, PB has a low mean completion time as well, suggesting that this mechanism is preferable over the rest. PBT, which performed well in the total completion time results, is undesirable in terms of the variation in completion times. Even though all four jobs are equally communication intensive, PBT could end up continuously boosting a single job in successive invocations of the periodic mechanism. In PB, this is avoided by checking message queues in a round-robin fashion. This effect can also be observed in fairness figures (Figure 2), where the periodic utilizations are imbalanced for PBT (the bars are not evenly split between the current jobs) compared to the four other schemes. These results suggest that for a homogeneous workload, PB is a good candidate to lower the completion times, has a low coefficient of variation of

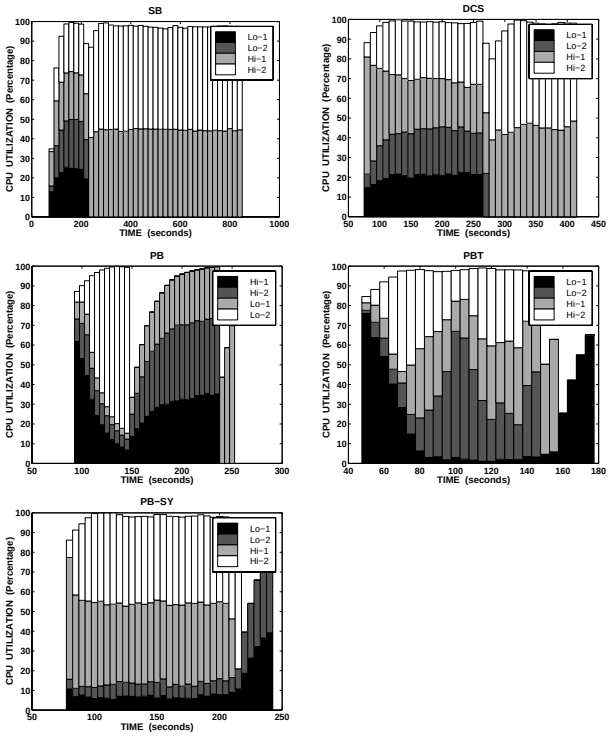

Figure 3: Monitoring CPU Utilization for Workload 3

completion times, and a reasonably equal split of CPU utilizations between current jobs.

Moving to the heterogeneous workload (Workload 3), we find PB, PBT and PB-SY are reasonable from the user's perspective in terms of the mean and coefficient of variation of completion times (Table 5). An examination of the fairness criteria (Figure 3) shows that SB and DCS give an equal share of the CPU to the current jobs in the system. PB-SY comes next and is fair to the extent that it does not totally starve out a process, but it still favors higher communication jobs. The fairness provided by PB and PBT is undesirable. DCS is a reasonable choice if both criteria are considered together.

3.3 Discussion

A clear lesson learnt from this exercise is that it is important to immediately schedule the destination process (if it is not currently scheduled) for which an incoming message is intended. This achieves two goals. It potentially schedules the destination at the same time as the sender of the message. It allows the destination to send back a reply to the sender (if needed) at the earliest so that the sender does not have to wait longer for the reply. Local scheduling does nothing in this regard, and hence performs poorly.

In Spin Block, the priority is boosted (in the interrupt service routine) only when the destination has blocked waiting for the message. However, if the destination has not yet arrived at the receive point (due to application skews), or even if it has arrived but has been context switched out in the middle of its spin, there is no immediate boost of its priority (to absorb the message). This seems to have a detrimental effect on the performance of Spin Block on

a programming model such as MPI, which has a coarser coupling between processes compared to a model such as the one used in [4]. In the implicit scheduling implementation on Split-C/Active Messages, a reply is sent back by the handler at the remote node in many cases. This tends to keep the sender and receiver more closely coupled, and as a result the blocking on a receive is expected to be more effective and is a better estimate of what is scheduled at the remote node. In MPI, the sends and receives are explicit and the effectiveness of our SB depends not just on message latencies and related overheads, but also on application work imbalance. This reiterates the need to study scheduling and communication jointly. One possible way of improving SB could be to keep a tighter coupling within the underlying MPI layer itself. For instance, we could transmit more flow-control messages (than strictly required) to implement the tighter coupling.

While Spin Yield seems attractive in terms of avoiding interrupt costs, the downside is that the priority boost for the destination is delayed even longer (since it occurs when another process at that node is ready to block). This suggests that Spin Yield should never be used in isolation. However, it can be used in conjunction with other schemes, such as Periodic Boost or Dynamic Coscheduling, that boost destination priority more often.

We find Periodic Boost consistently outperforming the other schemes for both high and low communication workloads. Periodic Boost is simple to implement since it does not require any additional functionality in the network interface or the user-level libraries. It does not add any overhead to the critical path of the message transfer mechanism either. Though not explicitly studied in this paper, it also offers the flexibility of employing more sophisticated heuristics (than just communication information) in scheduling decisions. Also, it can be used in conjunction with several other heuristics.

However, the "always schedule on arrival" strategy, mentioned above, is not without its pitfalls. It can have a significant impact on the variance of completion times and on the fairness to jobs. This could either lead to a job (which gets coscheduled first) holding on to the CPUs more than the others in a homogeneous workload, or could unfairly favor communication intensive jobs in a mixed workload. For homogeneous workloads, we find that PB is still a good candidate in terms of lowering the coefficient of variation of completion times as well as giving an equal share of the CPU to the current jobs. However, with heterogeneous workloads, PB is inadequate since it favors communication intensive jobs. SB and DCS are more fair under these circumstances. We find that PB-SY does not completely starve out low communication jobs (unlike PB) in a mixed workload, and does ensure that some progress is made though not equally. This is analogous to the traditional UNIX System V scheduler, which can unfairly favor I/O bound jobs in a mixture of CPU and I/O bound jobs though ensuring their individual progress. These results motivate the need for incorporating fairness criteria into the PB and PB-SY mechanisms, and we plan to explore this issue in our future work. It may be possible to use previously proposed schemes [20, 4, 3] for fairness (such as limiting the number of priority boosts for a particular process or periodically raising everyone to the highest level) in these mechanisms. In addition to these schemes, it is also possible to use current CPU utilizations in limiting the number of boosts that a process receives [20]. Since the PB function is executed independent of communication events, the scheduling decisions for fairness can also be taken at a different frequency than what is dictated by communication.

Finally one could argue that the coscheduling heuristics perform significantly worse than what one could achieve by just batching the jobs (or space sharing them if there are enough number of nodes). Batching, however, has the well-known problem of poor response times which is particularly disastrous for interactive jobs. With high-performance systems being increasingly used for graphics, visualization, databases and web services, in addition to the traditional large-scale scientific applications (short response times are important when debugging large scale applications as well), we believe that dynamic coscheduling strategies are a more viable option than batching. The disparity in results between these two options should rather serve as a motivation for future research in dynamic coscheduling strategies towards bridging this gap.

4 Concluding Remarks and Future Work

Efficient scheduling of processes on a NOW offers interesting challenges. Two operations, namely, waiting for a message and receipt of a message, can be used to to guide the system towards coscheduling without explicitly requiring extra explicit communication/synchronization . Independently combining the possible actions for each of these two operations, a 3×3 design space of scheduling strategies is obtained. Five of the these schemes are original contributions of this work, while the remaining four have been proposed in the past. This paper exhaustively evaluates the pros and cons of this design space through implementations on a uniform platform consisting of a network of Sun UltraSPARC workstations, running Solaris 2.5.1, connected by Myrinet. These schemes have been evaluated using varying mixes of several real parallel applications with different communication intensities. We find the newly proposed Periodic Boost (PB) scheme outperforming the others over a range of different workloads.

There are several interesting directions for future research to augment this study. Periodic Boost, which has proven very promising in this study, offers us the potential for several optimizations. We have only explored two possible heuristics. We may be able to use the number and/or recentness of outgoing messages, or even base it on actions totally unrelated to communication. We could dynamically control the frequency of invocation of the priority boost function, which has been statically set in this study. We also intend to examine the fairness issues in greater detail with regard to incorporating proportional fair share schedulers in conjunction with mechanisms such as PB. We plan to explore architecture and operating systems support that can improve the performance of these strategies, and help develop more efficient scheduling mechanisms to bring us closer to an ideal coscheduled environment, which is more scalable and reliable than a coscheduling implementation.

Acknowledgments

We would like to thank Morris Jette of Lawrence Livermore Labs (LLNL) for his comments and feedback on this work.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings*

of the ACM SIGMETRICS 1997 Conference on Measurement and Modeling of Computer Systems, pages 225–236, June 1997.

[2] R. H. Arpaci et al. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 267–278, 1995.

[3] A. C. Arpaci-Dusseau and D. E. Culler. Extending Proportional Share Scheduling to a Network of Workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1997.

[4] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, 1998.

[5] A. Banerjee. Implementation and Evaluation of MPI over Myrinet. Master's thesis, Dept. of Computer Science and Engineering, Penn State University, University Park, PA 16802, May 1999.

[6] N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] M. Buchanan and A. Chien. Coordinated Thread Scheduling for Workstation Clusters under Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[8] X. Du and X. Zhang. Coordinating parallel processes on Network of Workstations. *Journal of Parallel and Distributed Computing*, 46:125–135, 1997.

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

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

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] M. Lauria and A. Chien. MPI-FM : High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[13] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[14] Message Passing Interface (MPI) Forum. *Document for a standard Message Passing Interface*, 1994.

[15] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. An Experimental Study of Scheduling Strategies for a Network of Workstations. Technical Report CSE-98-009, Dept. of Computer Science and Engineering, The Pennsylvania State University, July 1998.

[16] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[17] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[18] E. W. Parsons and K. C. Sevcik. Coordinated Allocation of Memory and Processors in Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 57–67, June 1996.

[19] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[20] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1997.

[21] 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, April 1995.

[22] A. Tucker. *Efficient Scheduling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, November 1993.

[23] 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 15th ACM Symposium on Operating System Principles*, December 1995.

[24] 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 Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[25] F. Wong, D. Culler, and B. Saphir. MPI Performance on AM-II and GAM. http://now.CS.Berkeley.EDU/Fastcomm/MPI.

[26] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.