Re-examining Scheduling and Communication in Parallel Programs

Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler
{dusseau,remzi,culler}@cs.Berkeley.EDU

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract: **Modern MPPs and NOWs have evolved in ways that affect both the scheduling of parallel applications as well as the communication layer. The presence of a full operating systems upon each processor as well as the need to support interactive users substantially alter the traditional environment. Parallel applications may no longer be executing in the dedicated environment that fast communication layers, such as Active Messages, assume. In this paper we present a simulation-based study of the effects of a non-dedicated environment on parallel applications and investigate one method for reducing the resulting performance impact.**

Our results quantify the performance impact of the size of the flow-control window on parallel applications. We investigate increasing the size of this window to ameliorate the effect of various scheduling disturbances. Our results show that additional buffering in the communication layer significantly improves performance in the presence of large scheduling irregularities (e.g., those that occur when parallel applications are locally scheduled) but has a detrimental effect with smaller disturbances (e.g., quantum skew, daemon activity, and interactive users).

1. Introduction

In recent years, massively parallel processors (MPPs) and networks of workstations (NOWs) have been converging, with MPPs using the same processors, memory, and even operating systems as found in workstations, and NOWs beginning to use switch-based networks [ACP*95]. The workloads supported by the two environments are also converging: MPPs, such as the Intel Paragon [Gro92] and Meiko CS-2 [BCM94], are becoming more general purpose, supporting multiple users and acting as servers for sequential jobs as well as for parallel applications; NOWs are becoming capable of executing parallel applications in addition to sequential load sharing. In light of these potentially significant changes in the parallel computing environment, both the scheduling policies and communication layers used in traditional MPPs must be re-examined.

MPPs have classically approached the scheduling of parallel applications via one of two routes: batch-scheduling in conjunction with space-sharing or gang-scheduling. Both of these techniques provide applications with a dedicated virtual machine. However, the independent operating system running on each processor in a general-purpose MPP or a NOW may alter the execution environment. First, the underlying UNIX kernel on each processor could schedule threads of each parallel application independently. Second, even when gang-scheduling applications, context switches may not occur simultaneously in this more loosely-coupled environment. Third, daemon processes associated with the operating system must run periodically. These factors each invalidate the underlying assumption of a dedicated machine; in this study, we examine the impact on parallel application performance and investigate methods within the communication layer for reducing the impact.

Traditional MPP schedulers have ignored the difficulty of scheduling parallel applications in conjunction with short, sequential jobs requiring interactive response times. For example, in batch-scheduled, spaced-shared machines, a short job can be forced to wait until a resource-intensive application completes. Alternatively, systems like the CM-5 [Lei*92] time-slice parallel jobs within a partition with a reasonable time quantum to support short-lived parallel jobs; however, sequential jobs are not supported and each node executes only a primitive kernel with limited functionality. Time-slicing sequential jobs with parallel applications would maintain interactive response times, but wastes resources if parallel program performance is compromised or if processors sit idle. Future MPPs and NOWs must solve this scheduling problem to claim success as general-purpose platforms. In this paper, we begin to evaluate how parallel and sequential applications should share processor resources.

In a dedicated environment, the communication layer is given the guarantee that a message arriving at a processor is destined for the currently running process. This allows modern MPP communication layers, such as Active Messages [vEC*92], to achieve high performance by providing user-level communication and by avoiding message buffering. Even if parallel applications are gang-scheduled, a non-dedicated environment dictactes a number of changes to the communication layer. First, messages must be tagged with the destination process ID, and hardware or a trusted process must extract all packets from the network; messages which arrive for a non-scheduled process can no longer be handled immediately. Second, the view that clogging the network hurts only the parallel application sending the messages can not be retained. Finally, the entire network can not be drained of messages when switching between processes.

Two options exist for dealing with a message that arrives for a non-scheduled process: either discard the message (possibly notifying the sender of the scheduling discrepancy) or buffer the message until the destination process is scheduled. In this study, we examine the buffering approach and use credit-based flow control to manage the finite resources that the buffers present. Future work may include examining the discarding approach. The primary impact that this approach has on performance is that the sending process experiences additional latency on request-response messages that are buffered; the sender is stalled until the destination process is scheduled, the request message is handled, and the response is returned. Secondary performance costs arise from the overhead of performing the credit-based flow-control.

The theme explored throughout this paper is whether larger message buffers (and subsequently, larger flow-control windows) can lessen the performance degradation when messages arrive for non-scheduled processes. Split-C [CDG*93], the parallel language used by our applications, presents an attractive model for this study because it explicitly exposes the communication and synchronization dependencies between processes. Programmers can specify more relaxed assignment operators which may lessen the performance impact of a non-dedicated environment.

The rest of this paper is organized as follows. Section 2 presents our experimental environment, consisting of our programming language, our simulation methodology, and our benchmark applications. In Section 3 we explore the impact of the communication layer on parallel program performance. In Section 4, we compare the performance of gang-scheduling versus locally-scheduling parallel applications. We look at the performance impact of quanta skew in Section 5 and of daemon activity in Section 6. In Section 7 examine the effect of sequential jobs on parallel applications. We summarize our results in Section 8.

2. Experimental Environment

In this section we describe our experimental environment. We begin by presenting an overview of our programming language, Split-C. We then describe how we used a gang-scheduled MPP, the CM-5, to simulate scheduling policies and communication layers found in more general-purpose MPP or NOW environment. Finally, we briefly describe the benchmark applications used in our simulations.

2.1 Programming Language

Split-C is a simple parallel extension to C for programming distributed memory machines using a global address space abstraction [CDG*93]. It has been implemented on the CM-5, Paragon, SP-1, and various NOWs, using Active Messages to implement the global address space [Lun94,vEC*92,Mar94]. The language has the following salient features.

- A program is comprised of a thread of control on each processor from a single code image, i.e., SPMD.
- The threads interact through reads and writes on shared data, referenced by *global pointers* or *spread arrays*. These references require both a request and a response message.
- To allow the long latency of remote access to be masked by computation or issuing of other remote accesses, split-phase (or non-blocking) variants of read and write, called *get* and *put*, are provided. For example, given global pointer P and local variable X, X := *P initiates an access to the global address P. The operation is assumed to be incomplete until a *sync* statement is executed, which returns after all pending gets and puts complete.
- A form of write, called *store*, is provided to expose the efficiency of one-way communication in those algorithms where the communication pattern is known in advance. The threads can synchronize on the completion of a phase of stores or the recipient of stored values may wait for a specified amount of data.
- *Bulk transfer* within the global address space can be specified in any of the blocking or non-blocking forms.
- Threads may synchronize through global *barriers*.

Split-C exposes three distinct types of communication/synchronization dependencies among processes of a parallel application. The first and weakest dependency occurs when a processor *stores* data (i.e., sends a one-way message) to another process; this operation merely requires that messages are extracted from the network at the destination processor to prevent the sender from stalling. The second dependency is between a process *getting* or *putting* global data; these request operations require a response from the destination processor before the sender can complete the subsequent SYNC statement. Finally, the strongest dependency occurs when a process executes a *barrier*; in this case, the process must wait until all other processes have reached the synchronization point.

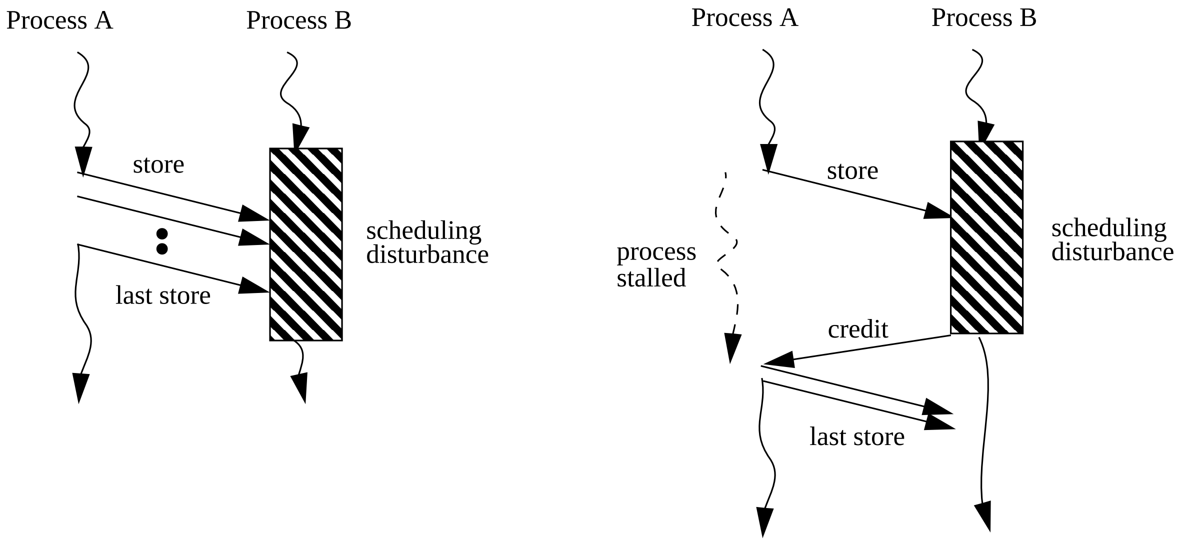

Figure 0: Impact of Scheduling Disturbance on Stores. *On the left, Process A repeatedly stores data to Process B. Because there is sufficient buffer space on the destination, Process A can continue to store data even during the scheduling disturbance. On the right, however, there is little buffer space on B. Thus, Process A is forced to stall until B is rescheduled and can respond with more credits. Thus, A feels the effects of the scheduling disturbance on the remote processor.*

The performance of a parallel application in a non-dedicated environment depends upon the prevalence of each of these three dependencies. Clearly, an embarrassingly parallel application has none of the three dependencies across processes and therefore is insensitive to scheduling disturbances. An application communicating only with store operations is insensitive to scheduling disturbances if two conditions are satisfied. First, to ensure that the sending process is not stalled, sufficient storage must exist to buffer messages that arrive when the destination process is not scheduled. Second, processes waiting to receive stored data must perform enough computation prior to the store synchronization point to mask the scheduling disturbances experienced by the sender. This situation is depicted in Figure 0.

In applications performing primarily *gets* and *puts*, only the requesting process can be stalled. For there to be no performance degradation in this case, not only must there be sufficient buffering at the destination, but the requesting process must perform a significant amount of computation between the issuing of the access and the *sync* statement. Only if the requesting process performs computation for the duration of the scheduling disturbance will it not be stalled.

Finally, applications which perform *barrier* operations frequently are the most sensitive to scheduling disturbances. A scheduling disturbance prevents a process from completing its work and progressing toward its synchronization point; therefore, other processes which have reached the *barrier* stall for an amount of time equal to the scheduling disturbance.

2.2 Direct Simulation

To measure the effects of scheduling disturbances on parallel applications, we use a technique called *direct simulation*. With direct simulation, a 64-node Thinking Machines' CM-5 [Lei*92] simulates a general-purpose MPP or NOW. We measure the execution time of one Split-C application gang-scheduled on the CM-5, while periodically interrupting each of the nodes to simulate other activity.

The mechanism for creating periodic, independent disturbances across the nodes is a user-level timer interrupt. By setting the interval at which the interrupt occurs, the duration of the timer handler, and the relationship across processors, we can simulate different scheduling policies and multi-program workloads. For example, to simulate a sequential application running on one processor while a parallel application runs on 64 processors, we activate the timer handler on only one processor. This "disturbance" appears to the parallel program as a sequential job time-slicing on one of the processors. By measuring the execution time of the parallel application, we can observe the slowdown induced by the simulated sequential application. This same mechanism is used to simulate a local scheduling policy for time-slicing between two parallel applications and for daemon activity.

To simulate scheduling policies that require coordination across processors, we use an asynchronous global interrupt; when one of the processors writes signals this interrupt, all processors simultaneously enter an interrupt handler. Measurements of handler start times across processors showed skews of less than 6 microseconds. In this manner we can simulate a gang-scheduling policy.

The similarity between our simulation environment and the system being measured allows us to easily model the costs associated with real systems. For example, context-switch overhead is modeled in our simulation environment as the overhead of executing the timer interrupt. Upon receipt of each timer interrupt, the operating system saves the integer registers, program counter, and other status registers, and vectors control to our user-level handler. At this point, we explicitly save the state of the floating-point registers. Similarly, by adding code to the timer handler that strides through memory, we are able to model cache effects. Note that we do not model network traffic originating from the simulated processes. The additional network traffic would increase the execution times of our applications due to both network congestion and the overhead of buffering additional messages.

The Split-C communication layer had to be modified to accommodate the non-dedicated environment. Namely, if an Active Message arrives for the application when the timer handler is executing, then the message can not be handled; instead the message is buffered and handled when the application is re-scheduled. We guarantee sufficient buffer space by using a credit-based flow-control message layer. A sender must now first check for credit availability on the destination process before sending. If there are not enough credits, the sender waits until outstanding messages (and credits) have returned. Our flow-control message layer builds on top of the CMAML [TM94] Active Message layer in the Split-C library; therefore, Split-C applications simply need to be recompiled to execute in this environment.

Our simulation methodology allows us flexibility over process scheduling, but unfortunately ties us to the network characteristics of our simulation testbed. In other words, our simulated MPP or NOW has the same packet size and network bandwidth and latency as the CM-5.

2.3 Benchmark Applications

We composed a suite of five parallel applications for our study, all written in Split-C. The applications were all tuned for the CM-5 and the original implementation of Split-C (i.e., they assume a dedicated environment). Our benchmarks represent a cross-section of parallel applications with different message sizes, communication dependencies (i.e., one-way versus request-response), and communication and synchronization frequencies. The characteristics of each benchmark determine its performance with different window sizes and scheduling disturbances.

The first benchmark, CHOLESKY, performs LU factorization on sparse, symmetric matrices. Our implementation relies primarily on *bulk get* and integer *store* communication operations. Its average message size is 70 bytes, and it communicates at a medium granularity, computing for several hundred microseconds between message sends. Implicit synchronization is performed frequently between pairs of processors.

CONNECT uses a randomized algorithm to find the connected components of an arbitrary graph [KLCY94]. CONNECT is built on single-packet one-way Active Message calls and integer *get* operations, so its average message size is a single CM-5 packet. CONNECT communicates at a very fine-granularity, computing on average only 40 microseconds between message sends.

COLUMN is an implementation of the column sort algorithm [Lei85]; a description of the implementation can be found in [CDMS94]. COLUMN is a bulk synchronous algorithm whose computation and communication phases last on the order of seconds; *barrier* operations are performed only at the end of each phase. It relies primarily on transpose operations for communication, storing very large messages (greater than 16K bytes) between pairs of processors.

The program EM3D simulates the propagation of electro-magnetic waves through objects in three dimensions [CDG*93]. EM3D communicates with *bulk store* operations, *stores* of integers and doubles, and both one-way and request-response active message calls, with about 100 us between communication events. Its average message is small, fitting into two CM-5 packets. The most important characteristic of EM3D is that it performs barrier synchronization frequently, more than one barrier every 10 ms.

Finally, SAMPLE[CDMS94] implements a sample sort [BLM91]. Like COLUMN, it is bulk synchronous, performing only a few barrier synchronizations and consisting of communication and computation phases lasting many seconds. In the communication phase, each processor sends many (128 K) one-way single-packet active messages to random destination processors

3. Dedicated-Environment Window Size

Before exploring the effect of a non-dedicated environment on parallel applications, we examine the impact of our communication layer. To determine the optimal message-layer window size in a dedicated environment, we ran each of our Split-C applications with no scheduling disturbances using window sizes between one and 512 packets, where the window size is the number of packets that can be sent to each destination process before receiving additional credits. Figure 1 shows that the execution time of three of the applications is extremely sensitive to window size; SAMPLE and EM3D are 60% faster when the window size is chosen correctly; COLUMN is 20% faster. The slowdowns are shown relative to the optimal window size, not to the original Split-C version. SAMPLE with our flow-control layer is twice as fast as the CMAML version; the other applications are

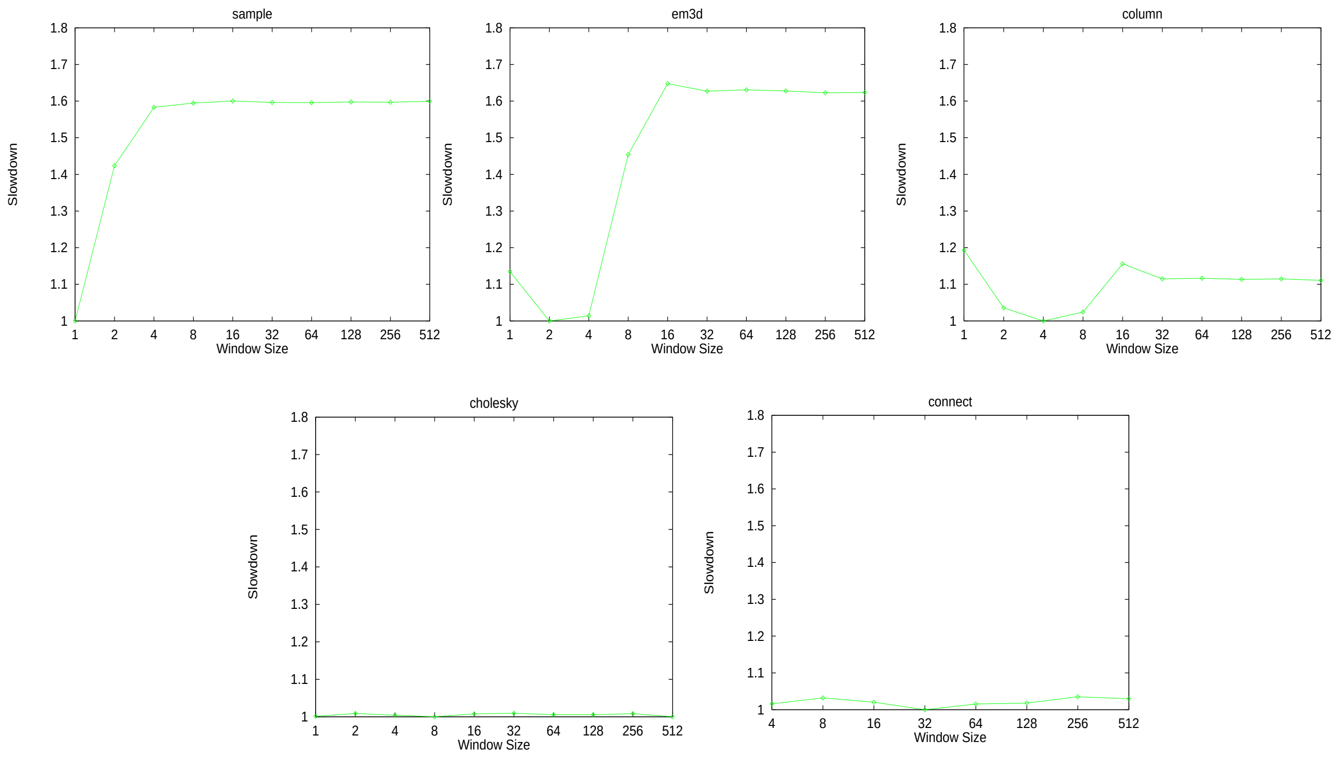

Figure 1: Sensitivity to Window Size. *This figure shows the slowdown of each application as a function of the communication layer window size (in numbers of packets) on a 64-node CM-5. The slowdown is the ratio of the execution time with the current window size to the execution time with the optimal window size. Each data point represents the mean of five measurements.*

10% to 40% slower with our modified communication layer.

The execution time of SAMPLE is minimized when the window is set to one packet due to its communication phase involving single-packet one-way messages to random destinations. This communication phase incurs contention at random processors, with the hot spots in the network moving to different processors throughout the phase. If each sender is forced to wait until the previous message to that destination has been acknowledged, then no processor can be flooded with messages. EM3D also benefits from a small window size because it primarily communicates with short one-way stores to random processors; since the average amount of data sent fits into two packets, a window size of two has better performance than a window of one.

COLUMN's communication phase consists of very large bulk stores between pairs of processors; in this case, the best performance is achieved when the window size is slightly less than the round-trip time of the network. A smaller than optimal window forces the sending processor to stall on acknowledgments from previous stores, preventing stores from being fully pipelined and wasting network bandwidth. With larger windows, the send rates between pairs of processors are not kept in balance. If one processor sends faster than its partner, then the slower processor is forced to handle the arriving messages before sending out its own; however, if the sending processor experiences a slight delay between sets of message sends, then the slower processor can send in that time and the pair is kept balanced. Since the time of this phase is the time of the slowest processor, optimal performance occurs when processors complete simultaneously.

Neither CHOLESKY nor CONNECT are sensitive to the size of the window, due to the fact that they communicate primarily with request-response messages of a small size. Requiring a response to a message has a performance impact similar to a window size fixed at the average message size.

In summary, we see that a subtle change in the communication layer can have a noticeable impact on parallel application performance. The choice of window size can sway performance results up to 80%. Further, we expect that the network in a NOW will support larger packets and have longer latencies. Therefore, programs that are sensitive to the depth of the network or packet size (e.g. COLUMN and EM3D, respectively) will require larger windows for optimal performance.

4. Coscheduling Parallel Applications

Coscheduling [Ous82] and *gang scheduling* [GTU91] are two similar, traditional techniques for scheduling parallel applications on MPPs. Both operate under the principle that higher performance is achieved when all processes of the same parallel application are scheduled simultaneously. We distinguish between the two scheduling policies by recognizing that coscheduling is less strict than gang scheduling. With coscheduling, simultaneously scheduling parallel processes is beneficial for performance, but it is not necessary for correctness. On the other hand, gang scheduling requires that parallel processes are scheduled simultaneously for correctness, guaranteeing that messages can not arrive from other processes. By these definitions, parallel applications on a NOW or modern MPP may be coscheduled, but not gang scheduled, since multiple processes can be simultaneously running on different processors and no mechanisms exist for preventing processes outside the cluster from communicating with any processor within.

For a comparison point, we measured the execution time of our parallel applications in a simulated coscheduled environment where each benchmark is time-sliced with a simulated parallel job with a quantum of 100 ms. The costs of the context-switch and flushing the cache between time-slices are included in our measurements. As expected, total execution time is within five percent of the sum of the dedicated run times, and the optimal window sizes are identical. Note that cache flushing with a quantum of 100 ms was found to have a negligible effect.

Scheduling policies which do not require global coordination may be an attractive alternative in systems with a complete operating system on each node, due to scalability or fault-tolerance constraints. One approach to scheduling parallel jobs is to allow the underlying UNIX kernel on each processor schedule the parallel applications independently; we term this *local-scheduling*. This policy, employed by parallel environments such as PVM [Sun90], has the advantage that no kernel modifications are required; however, as has been shown in previous studies [FR92], local scheduling leads to unacceptable execution times for processes that communicate frequently. We confirm this result for our five benchmark applications while investigating the impact of window sizes on performance.

Figure 2 shows the slowdown of each application when locally-scheduled with one other (simulated) parallel job with time quanta of 100 ms and 500 ms. Results in the graph on the left are with the window sizes that are optimal in the dedicated environment; results in the graph on the right are with the optimal window sizes for this environment. With a time quantum of 100 ms, the parallel applications require up to seven times more CPU time than in a dedicated environment; changing the window size does not improve performance significantly. EM3D is affected the most severely because of its frequent barrier operations. Surprisingly, SAMPLE, CHOLESKY, and CONNECT are all slowed down by less than 90%. This slowdown may be acceptable considering that no global scheduling is required; however, as more than two parallel applications are time-sliced on the same workstations, slowdowns increase substantially.

Performance degrades severely when the time quantum is increased to 500 ms, because processes must wait longer on dependencies with non-scheduled processes; slowdowns range between 5 and 75. Increasing the window size dramatically improves performance if the dependency between processes was simply a one-way message, i.e., the sender stalled for window credits. Larger windows remove the artificial dependency of processes waiting for buffer space; processes now stall only for true dependencies, i.e., when they need a response to a get or put request or when waiting on a barrier. However, some applications are still slowed down by more than a factor of 20.

In conclusion, some form of global coordination is required for parallel application performance, due to dependencies

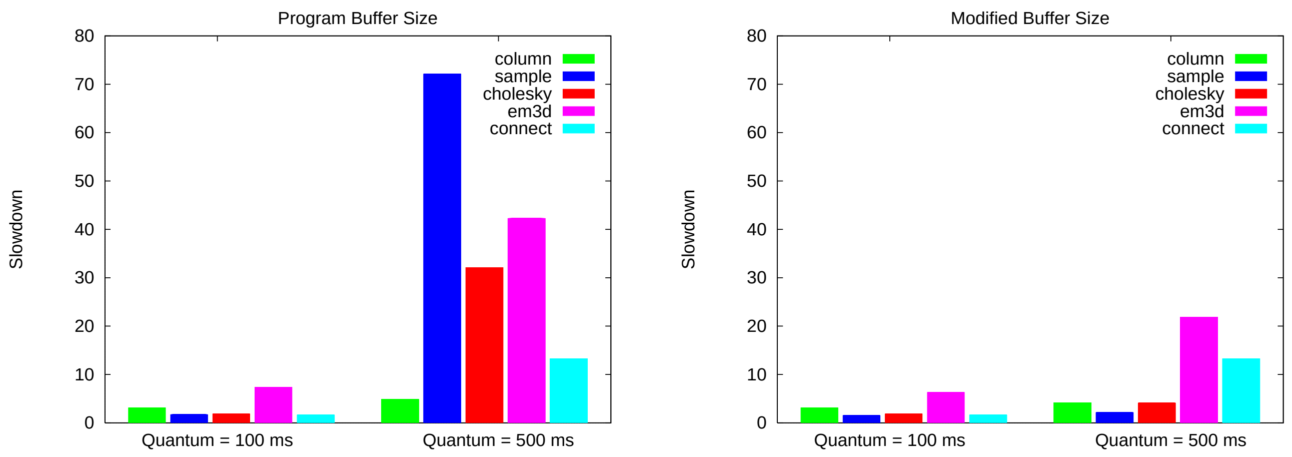

Figure 2: Locally Scheduled. *The leftmost figure shows the slowdown of the applications when locally-scheduled with one other parallel application, using the window sizes that were optimal running in dedicated-mode. The rightmost figure uses window sizes that are optimal when locally-scheduled. Results are shown for 100 and 500 ms time quanta. Slowdown is the ratio of the locally-scheduled execution time to the coscheduled execution time.*

across processes. In the next two experiments, we examine the effects of slight scheduling disturbances on global coordination.

5. Coscheduling Skew

In the previous section, we assumed perfect coscheduling---that is, context switches across workstations began and ended at precisely the same moment. Such coscheduling may be possible in an tightly-coupled MPP because a common clock is distributed to all nodes. However, in a physically distributed system, perfect coscheduling is unlikely to be practical. For example, if context switches are coordinated by broadcasting a signal from a master node, different processors may receive the signal at different times. Even if coordination exists because the clocks of the workstations have been synchronized [Lam90], it is likely that some clock skew exists across workstations. Finally, context switches may even take different amounts of time due to cache effects.

We define the coscheduling skew time as the maximum difference between the quantum start times across processors. To quantify the effect of coscheduling skew on performance, we offset the start time of each processor’s time quanta by a random amount, up to 10 ms. Our simulation results, shown in Figure 3, are somewhat noisy because we are measuring slowdowns close to the variation across experimental trials. With a 100 ms time quantum, all of our applications see an increase in execution time as the skew increases; however, the slowdowns are all relatively small: EM3D experiences the most significant slowdown of 5% when the skew is 10% of the quantum length. We see that increasing the time quantum to 1 second has the result that slowdown remains below 3% for all data points, with no real increasing trends in slowdown as the skew increases.

We found that increasing the window size did not affect our results; since the program is coscheduled most of the time, it performs best with the dedicated buffer size. That is, even if a larger buffer size is beneficial when the processes are not perfectly coscheduled, the larger buffer size hurts the performance for the majority of time when the application is coscheduled. In conclusion, we discovered that with a large time quantum (1 second) or small skew times (less than 10 ms), parallel applications are minimally affected by imperfect coscheduling, exhibiting slowdowns of less than 5%.

6. Daemon Activity

Processors with complete operating systems have daemon processes which must run periodically. Table 1 shows measure-

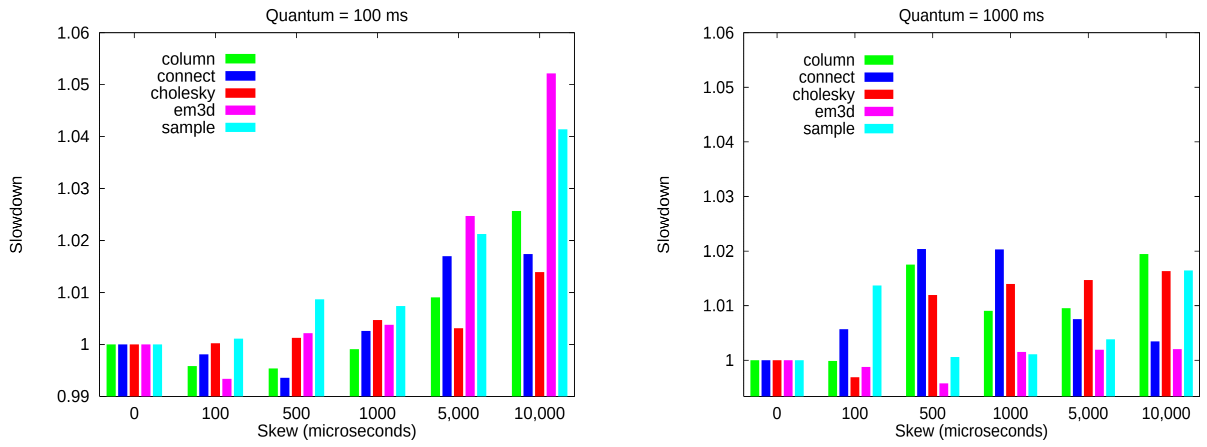

Figure 3: Coscheduling Skew. *Each parallel application is coscheduled with another application, but the time quanta are skewed across processors. The skew along the x-axis is the maximum difference in start times across processors; it is varied between 10 us and 10 ms. The slowdown is relative to the perfectly coscheduled performance.*

ments of the average run time and inter-arrival times for the top running daemons on a DECstation 5000 running Ultrix 4.2a. These measurements indicate that a daemon process runs approximately once every 10 seconds for about 5 milliseconds on each workstation.

Daemon	Run Time (ms)	Interval (s)
cron	3 to 6	60
update	3 to 5	30
routed	1 to 2	25
Xdec	1	25
init	1	200

Table 1: **Daemon Activity.** *This table shows the average execution times and intervals between invocations of the top daemon processes as measured over a three hour period when the machine was idle of all user activity. These measurements were performed by Doug P. Ghormley.*

We simulate the effect of daemons arriving independently across processors at intervals between 5 and 30 seconds for durations of 5, 10, and 50 ms. In our first set of experiments, we assume that the daemon process periodically polls the network and that the messages for the parallel application are buffered; however, the daemon process has a fixed amount of work to accomplish and as a result may run longer if it must also buffer messages. In our second set of experiments, the daemons ignore the network.

Our results in Figure 4 show that with interruptions characteristic of current daemon activity, parallel applications are slowed down by less than 10%, regardless of whether or not daemons poll the network. Daemons of a slightly longer duration (e.g. 10 ms) arriving more frequently (e.g. every 5 seconds) slow down the applications by less than 20%, with the exception of CONNECT, which exhibits one data point with an 80% slowdown.

If parallel applications are scheduled independently of longer system activity, such as 50 ms durations, then large slowdowns are observed for most applications. The slowdown is surprising, since even these long-running daemons are scheduled less than 1% of the time on each processor. Once again, EM3D is slowed down the most (2.4 times) by scheduling perturbations because of its frequent barrier points. The applications SAMPLE and CONNECT are also strongly affected by daemon activity, since their communication patterns have little locality; a process must wait before sending when the destination processor

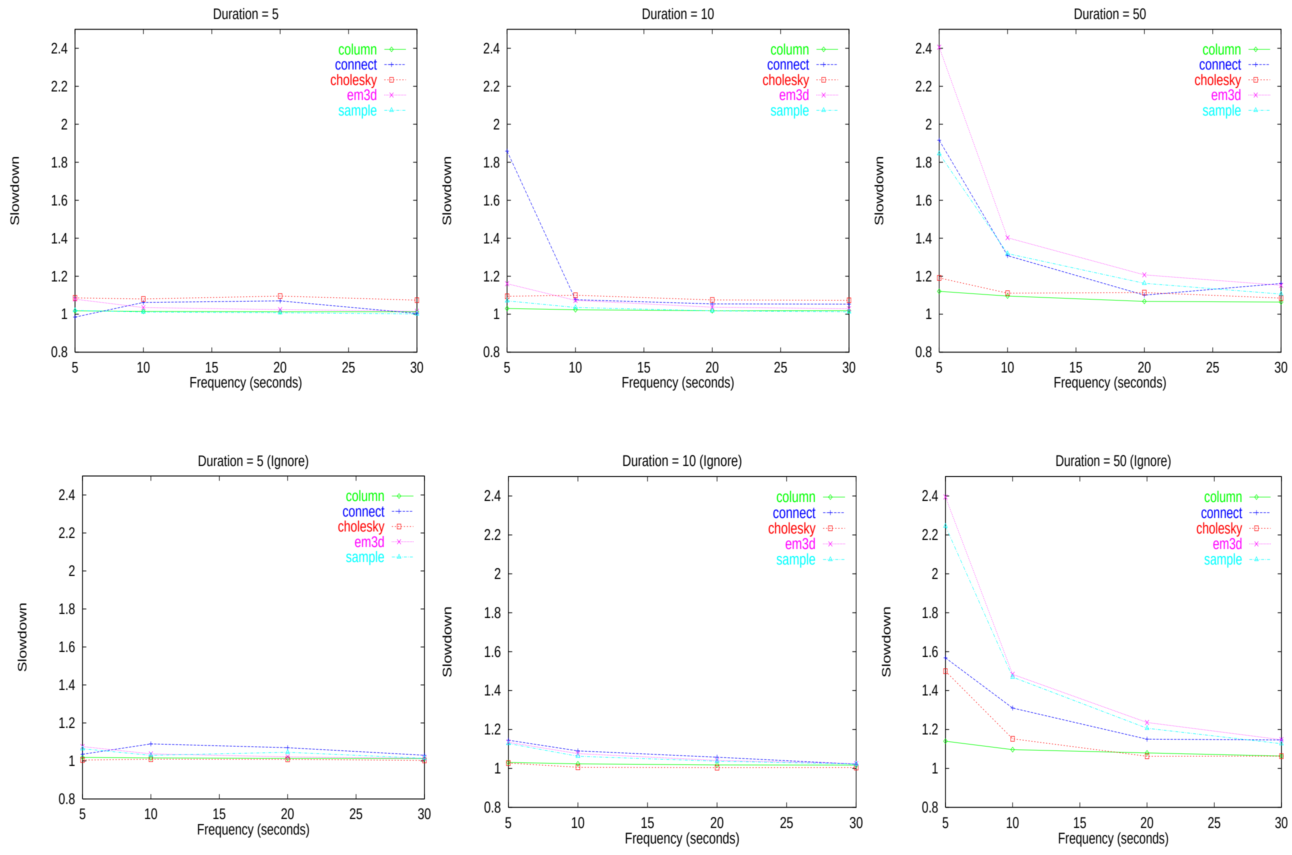

Figure 4: Daemon processes. *The duration of the daemon processes is set to 5, 10, and 50 ms in successive graphs, while the inter-arrival time is varied along the x-axis from 5 to 30 seconds. The slowdown is the ratio of the measured execution time to the execution time of the application running in a dedicated environment. In the top three graphs, the daemon process periodically polls the network and buffers messages for the parallel application. In the bottom three graphs, the daemon process ignores the network.*

has a daemon scheduled, and then has very little to send to that processor. COLUMN suffers less than 20% because it synchronizes infrequently and communicates with large messages; therefore, although a process sometimes stalls before sending, once the destination process is scheduled, the sender has many packets to send and is not affected by the daemons running on the other workstations.

As was the case with coscheduling skew, increasing the buffer size does not improve performance with daemon activity. Because daemon activity occurs relatively infrequently, processes communicate with scheduled processes for the majority of the time and experience optimal performance in this situation with small window sizes. An interesting experiment would be to dynamically change the operating window size depending upon whether or not the destination process is currently scheduled to see if this brings a performance advantage.

With current levels of daemon activity (namely, an interval of 10 seconds and a duration of 5 ms), parallel applications suffer a noticeable 2 to 10 percent slowdown. This slowdown can probably be ignored, but increased system behavior (ironically, perhaps daemons that monitor system activity to find idle workstations in a NOW) might require a more drastic solution, such as coscheduling daemon activity across processors.

7. Scheduling with Interactive Jobs

Traditional MPP schedulers assume that parallel applications are the only existing jobs. This is not a fair assumption in NOWs or in emerging MPPs, which are capable of acting as servers for sequential jobs. Studies have shown that a NOW cluster can sustain a “typical” workload of parallel applications, assuming that each application requires only half as many processors as there are existing workstations[ADV*94]. Thus, as long as parallel applications do not request more than this number of processors, enough resources exist to ensure that parallel processes can be migrated to an idle workstation if a user reclaims their workstations. However, if parallel applications require as many processors as exist in the system, then clearly, processors must be shared between parallel and sequential applications.

We simulate sharing one non-idle workstation between an interactive job and a parallel application and determine the impact on the parallel application. The parallel application runs 100% of the time on the 63 idle processors and is time-sliced in a round-robin fashion with the sequential application on the one non-idle processor. We vary the percentage of time that the parallel process executes on the non-idle node between 20% and 100% and vary whether or not the sequential process polls the network. Figure 5 shows the results for this experiment, with a quantum of 100 milliseconds.

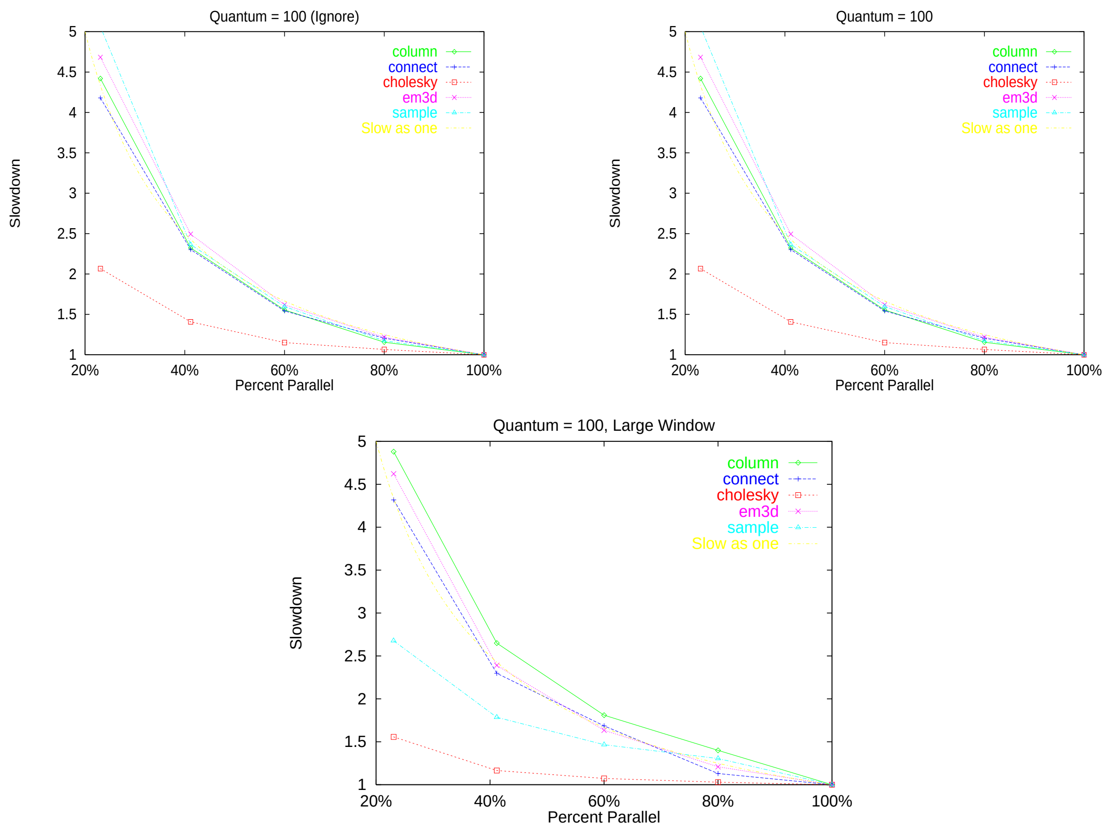

Figure 5: One Non-Idle Processor. *A serial job is time-sliced on one processor with a parallel application running on 64 otherwise idle processors. In the first graph, the serial job ignores the network; in the second, it periodically polls and buffers messages. The third graph shows that increasing the flow-control window improves the performance of two applications. The serial job is always given a time-slice of 100 ms. The percentage of time the parallel application is scheduled is varied from 20% to 100% along the x-axis. The slowdown is the ratio of the measured execution time with one non-idle workstation*

If the work in a parallel application is evenly distributed across processes or if barrier synchronizations are frequently performed, and one process is scheduled less frequently, then the entire application is slowed down to the rate of the slowest process; this is the case for two of our applications: EM3D, and COLUMN. On the other hand, if a load-imbalance exist across processors and a processor with less work than others is interrupted, then the entire application is not slowed down by the same amount. This is the case for CHOLESKY and CONNECT. A load-imbalance also exists in SAMPLE, but the interrupted processor was already one of the slowest and so the performance curve follows this processor. [1]

Our results in the second graph indicate that whether the sequential process buffers messages or ignores the network has little impact on performance. Experiments varying the window size in the third graph show that performance is improved for two of the applications.. The performance of SAMPLE improves dramatically with a larger window because processes no longer stall after sending a one-way message to the non-idle processor. CHOLESKY also sees a further improvement. In the other three applications (COLUMN, CONNECT, and EM3D), it may still be beneficial to use a larger window when sending to the non-idle processor; however, because the same window size is used for communicating with all processes and most of the time processes send to idle processors, better overall performance is achieved using the smaller window sizes.

Our experiments have shown that parallel applications are generally slowed to the rate of the slowest processor when shared with an interactive user, regardless of whether the serial job interacts with the network. The resulting implications depend upon the execution environment. In a NOW with sufficient idle resources, parallel applications should be migrated away from busy workstations[2]. However, in environments where resources are over-utilized or when applications require the maximum number of processors, migration is not an option and processors must be shared. Since little benefit exists in running fragments of a parallel application, other processes should be scheduled on the available processors when the sequential application runs. A related question, which we plan to pursue, is at what point does the need to coschedule sequential jobs arise. Our experiments have shown that with one sequential job and one parallel job, there is no need to explicitly coschedule the parallel application since the parallel job performs no worse than the slowest process.

8. Related Work

There have been numerous studies on multiprocessor scheduling techniques. Almost all of these, however, have been based on shared-memory architectures or have used synthetic inputs to drive their simulations. We focus on real parallel programs in a distributed-memory environment, which we think will typify a NOW.

Ousterhout first introduced the idea of *coscheduling*[Ous82]. The idea has since been included in many studies of multiprocessor scheduling techniques [TG89, CDD*91, FR92, LV90]. Ousterhout assumed that coscheduling is a good idea, and implemented and compared three algorithms, finding the matrix algorithm simplest and nearly most efficient. Gupta et al. [GTU91] found that coscheduling was as good as their space-sharing method (known as *process contro*l), if the time quantum was long enough (25 milliseconds) to amortize cache effects.

Others have studied how to determine which processors and how many processors to allocate to a parallel application [Sev89, NSS93, CMK94, MZ94]. We assume that the application will demand and must be given a certain number of processors.

In another study of mixing parallel programs into a workstation cluster, simulation is used to study whether parallel applications can run in a non-dedicated environment (such as a NOW) [LS93]. While demonstrating that this may be possible without a significant impact on the parallel programs, their work does not discuss any potential impact upon interactive users. Further, all the simulations are driven by synthetic models of both workstation and parallel program behavior. Finally,

1. Our measurements are reported as the mean of five repetitions; however, in every experimental run, we run the sequential job on the same processor. When a load-imbalance exists across processors, it would be interesting to look at the effect of slowing down different processes in the application; we plan on performing this experiment in future work.
2. Our experiments were performed on applications that assume a fixed number of processors. Applications that dynamically allocate work across processors (such as with a task queue) or that dynamically adapt the amount of parallelism to the number of available processors may not need to migrate

[ADV*95] performed a similar, but more limited study in the NOW environment, that did not examine the effects of changes in the communication layer on application performance.

9. Conclusions

Modern MPPs and NOWs have evolved in a number of ways that impact both the scheduling and the communication layer of parallel applications. Interactive users, local scheduling, quantum skew, and system activity all lead to scheduling perturbations in parallel applications. The result is that parallel applications are no longer strictly gang-scheduled and the communication layer must be modified to take this into account. One modification, which we examined in this paper, is to buffer messages that arrive for non-scheduled processes, using credit-based flow-control to ensure sufficient buffer space.

We have found that in situations where the scheduling disturbances are large (e.g., when parallel applications must share a workstation with an interactive job and when parallel applications are locally scheduled), the degradation on parallel application performance is substantial. Increasing the amount of buffer space for messages improves the performance of applications which communicate primarily with one-way messages, but better performance is ultimately obtained by coscheduling these applications, in which case sensitivity to window size is just as in a dedicated environment. Thus, if few sequential processes are present, they should be migrated to idle processors apart from the parallel program, or if a large number of sequential processes exist they can be coscheduled.

When the scheduling perturbations are small (e.g., due to coscheduling skew or daemon activity), we have found that larger message buffers do not improve performance for two reasons. First, data dependencies and synchronization points in applications necessitate that messages are handled and responses returned; providing more buffer space does not change this fact. Second, the performance of coscheduled processes can be extremely sensitive to window size; in these cases, we found that the optimal window size is small and can lead to an 80% performance improvement over larger window sizes. Therefore, given small scheduling perturbations, the best performance is achieved again with the window size that is optimal when coscheduled.

10.References

[ACP*95] T. Anderson, D. Culler, D. Patterson, et.al., “A Case for NOW (Networks of Workstations),” *IEEE Micro*, February 1995.

[ADV*95] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson, “The Interaction of Parallel and Sequential Workloads on a Network of Workstations.” To appear in *ACM SIGMETRICS and Performance ‘95*, May 1995.

[BCM94] E. Barton, J. Cownie, and M. McLaren, “Message Passing on the Meiko CS-2”. *Parallel Computing*, 20(4):497-507. Apr 1994.

[BLM91] G. Blelloch, C. Leiserson, and B. Maggs, “A Comparison of Sorting Algorithms on the Connection Machine CM-2”, *Symposium on Parallel Algorithms and Architectures*, July 1991.

[CDG*93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, “Parallel Programming in Split-C,” *Proceedings of Supercomputing ‘93*, 1993.

[CDD*91] Crovella, M., Das, P., Dubnicki, C., LeBlanc, T. and Markatos, E., “Multiprogramming on Multiprocessors”, University of Rochester, Computer Science Department, February 1991, num. 385.

[CDMS94] D. Culler, A. Dusseau, R. Martin, and K. Schauser, “Fast Parallel Sorting under LogP: from Theory to Practice,” pages 71-98, *Portability and Performance for Parallel Processing*, John Wiley & Sons Ltd., 1994.

[CMK94] Chiang, S., Mansharamani, R.K. and Vernon, M.K., “Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies”, *Proceedings of the 1994 ACM SIGMETRICS Conference*, 33-44, February, 1994.

[FR92] D. Feitelson and L. Rudolph, “Gang Scheduling Performance Benefits for Fine-Grained Synchronization,” *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pages 306-318, Dec., 1992.

[Gro92] W. Groscup, “The Intel Paragon XP/S Supercomputer.”, In *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology.*, pages 262-273, Portland, Oregon, Nov. 1993.

[GTU91] A. Gupta, A. Tucker, and S. Urushibara, “The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications,” *Proceedings of the ACM Sigmetrics Conference*, May, 1991.

[KLCY94] A. Krishnamurthy, S. Lumetta, D. Culler, and K. Yelick, “Connected Components on Distributed Memory Machines,” The *Third DIMACS International Algorithm Implementation Challenge*, 1994.

[Sev89] Sevcik, K.C., “Characterizations of Parallelism in Applications and their Use in Scheduling”, *Proceedings of the 1989 ACM SIGMETRICS Conference*, 171-180, May, 1989.

[Sun90] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing,” *Concurrency: Practice and Experience,* vol. 2, no. 4, pages 315-339. Dec, 1990.

[Lam90] L. Lamport, “Concurrent Reading and Writing of Clocks,” *ACM Transactions on Computer Systems*, vol. 8, no. 4, pages 305-310, April, 1990.

[Lei*92] C. Leiserson, et. al. “The Network Architecture of the Connection Machine CM-5,” *Symposium on Parallel Algorithms and Architectures*, April 1992.

[Lei85] T. Leighton, “Tight Bounds on the Complexity of Parallel Sorting”, *IEEE Transactions on Computers*, April, 1995.

[LS93] Leutenegger, S.T. and Sun, X., “Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System”, *Proceedings of Supercomputing 93*, November, 1993.

[LV90] Leutenegger, S.T. and Vernon, M.K., “The Performance of Multiprogrammed Multiprocessor Scheduling Policies”, *Proceedings of the 1990 ACM SIGMETRICS Conference*, 226-36, May, 1990.

[Lun94] S. Luna, “Implementing an Efficient Global Memory Portability Layer on Distributed Memory Multiprocessors,” Masters Thesis, University of California, Berkeley, May, 1994.

[Mar94] R. Martin, “HPAM: An Active Message Layer for a Network of HP Workstations”, *Hot Interconnects II,* August, 1994.

[MZ94] McCann, C. and Zahorjan, J., “Processor Allocation Policies for Message-Passing Parallel Computers”, *Proceedings of the 1994 ACM SIGMETRICS Conference,* 19-32, February, 1994.

[NSS93] Naik, V.K., Setia, S.K., and Squillante, M.S., “Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments”, *Proceedings of Supercomputing 93*, 824-833, November, 1993.

[Ous82] J. Ousterhout, “Scheduling Techniques for Concurrent Systems,” pages 22-30, *Third International Conference on Distributed Computing Systems*, May, 1982.

[TG89] Tucker, A. and Gupta, A., “Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors”, *Operating Systems Review,* 1989, vol. 23, num. 5, 159-66.

[TM94] L. Tucker and A. Mainwaring, “CMMD: Active Messages on the CM-5”, *Parallel Computing,* 20(4):481-496. Apr 1994.

[vEC*92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, “Active Messages: A Mechanism for Integrated Communication and Computation,” *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992.