Barrier Optimizations in Implicit Coscheduling

Michael J. Brim, Todd L. Miller
Computer Sciences Department
University of Wisconsin-Madison
{mjbrim,tlmiller}@cs.wisc.edu

18th December 2001

Abstract

Implicit coscheduling allows member processes in a parallel application to remain coordinated, reducing overall execution time when run on time-shared distributed systems such as networks of workstation (NOWs). The existing implementation of implicit coscheduling performs close to or better than the ideal, gang scheduling, for a variety of application classes, including bulk-synchronous, continuous-communication, and load-imbalanced applications. Still, there is one class of applications that does not benefit from the strategies of implicit coscheduling, that class being continuous-communication applications where the communication interval is small, the synchronization interval is large, and the load-imbalance within the application is large. We attribute the bad performance of this class of applications to the poor choices made by local schedulers during barrier operations, as it can be shown that synchronization time accounts for approximately 30% of the total execution time for these applications, even though barriers are infrequent. As such, we present multiple techniques for improving barrier performance for this class of applications. First, we investigate the use of tree-based processing of barrier messages, in order to distribute messaging overhead and overlap communication. Next, we present several alternative local waiting algorithms that make use of information gathered by several methods. Finally, we describe an algorithm for explicit synchronization of processes at barriers using keep-alive messages sent from the barrier's root process. Our results indicate that most of the optimizations presented either only slightly improve performance or do not provide any significant improvement, suggesting that it is inherently difficult to predict the actions of local processes during barriers in such applications.

1 Introduction

The current trend in both industry and research institutions to use networks of workstations (NOWs) as general purpose compute servers for sequential and parallel applications [1] [12] has led to the problem of efficiently time-sharing the processors in the NOW between applications without degrading the performance of either type of application. The problem stems from the performance degradation encountered by parallel applications when leaving scheduling decisions in the hands of the commodity operating systems local to each workstation. Since the local operating system scheduler has no knowledge of the communication patterns of parallel applications, it may make scheduling decisions that cause processes to block when in fact they should remain scheduled so as to allow its coprocesses to continue. In order to meet such performance goals, much research has been performed in efforts to develop efficient methods of coscheduling parallel applications on NOWs.

In [3], the authors examine the different approaches to designing a distributed system, and propose the use of an information-based approach to such designs. Specifically, they identify four common methods for obtaining the information required by the components in a distributed system to adapt to current conditions. These methods include: a) null, or no information gathering; b) implicit, where information is gathered only through observation of current system conditions; c) parasitic, where additional state information is piggybacked on top of existing communication; and d) explicit, where components explicitly query other components to determine the system state. Each of the previous attempts to develop an efficient mechanism for coscheduling parallel applications can be categorized as one of these four methods. In our work, constraining information gathering to one of these categories seemed limiting. As such, our approaches make use of implicit, parasitic, and explicit methods of information sharing in order to improve upon the current strategies employed by implicit coscheduling.

As described in [4], implicit coscheduling allows member processes in a parallel application to remain coordinated, reducing overall execution time when run on time-shared distributed systems. By observing messages from other coprocesses, a process can implicitly infer the state of those processes and take appropriate action to balance local processor efficiency with parallel application performance. The actions taken by local processes under implicit coscheduling are determined through the use of conditional two-phase waiting. The idea behind conditional two-phase waiting is that processes always spin wait for a base amount of time in order to remain coordinated during the current operation, and then conditionally wait longer if it can be inferred that doing so will improve the performance of the application. An example of when a process should conditionally spin is when it has received messages from other processes recently that were not related to the current operation, indicating that if the process were to block that it may cause other processes being served by the process to block as well.

Implicit coscheduling does not require changes to the operating system, as all support is provided in the Split-C parallel programming library [5] and its underlying communication mechanism, Active Messages [11]. Split-C is a parallel extension to the C language that provides a global shared address space and a set of simple remote access operations that allow cooperation of processes on distributed memory multiprocessors. Split-C aims to gain all the benefits of shared-memory, message-passing, and data parallel programming while minimizing the detrimental effects posed by each. Active Messages is an asynchronous communication mechanism developed to maximize the performance of communication on message-passing multiprocessor machines. Active messages employs a simple strategy where each message contains the location of a user-level message handler that is invoked upon message reception and is responsible for retrieving the message from the network layer and integrating it into the ongoing parallel computation. By allowing overlapping of computation and communication and fully exploiting the available hardware for processing messages, Active message achieves the desired performance level needed by current high-performance applications.

Implicit coscheduling performs well for a wide variety of application classes, including bulk-synchronous and continuous communication. In environments where applications perform barrier synchronization often and communication is coarse-grained, implicit scheduling actually outperforms the form of scheduling it uses as an ideal, gang scheduling. However, the ability of processes to determine the correct action during a barrier is affected when the application contains load-imbalance, since processes are unable to differentiate between whether a coprocess is unscheduled or the application contains load-imbalance using only implicit information obtained from observing message arrivals. With incorrect information, a process may unwisely choose to block when it would be more beneficial to continue spin waiting or vice-versa. In most cases, implicit coscheduling still performs reasonably well in the presence of load imbalance, showing limited slowdowns of only 20-40% versus gang scheduling. There is one glaring exception, however, as reported in [4]. For continuous-communication applications where the communication interval is very small (10us), the synchronization interval is very large (100ms), and the load imbalance factor is high (1.5), implicit coscheduling shows a 70% slowdown over the ideal.

Our work focuses on this last class of applications, hoping to improve the performance of implicit coscheduling for these applications to the values shown by other classes. Through analysis of the simulation of such applications, it can be seen that although synchronization events are infrequent, synchronization accounts for approximately 30% of the total execution time independent of the length of the jobs. As a result, our approaches to improving the performance of implicit coscheduling for this class of applications will be focused on improving the performance of the time consuming barrier synchronization operations through optimization and extension of the current algorithms.

The rest of the paper is organized as follows. Section 2 describes our work in the context of others with the same general goals. Section 3.1 summarizes our methodology and research environment. Sections 3.2 to 3.4 present our approaches for optimizing the performance of barrier operations, along with the results for each approach. In 3.2, we investigate the use of tree-based barrier messaging. In 3.3, an investigation is made into alternate waiting algorithms for local spin decisions. In 3.4, the impacts of explicit keep-alive messaging are presented. Finally, in section 4, we conclude and report unresolved issues and other areas for future work.

2 Related Work

The work presented is an extension to the existing design of implicit coscheduling [4], which performs well in environments where applications use barrier synchronization often and communication is coarse-grained. However, the ability of processes to determine the correct action during a barrier is affected when the application contains load-imbalance, since processes are unable to differentiate between whether a coprocess is unscheduled or the application contains load-imbalance using only implicit information obtained from observing message arrivals. With incorrect information, a process may unwisely choose to continue spin waiting when it would be more beneficial to block or vice-versa. Our work focuses on extending the implicit coscheduling work to eliminate this limitation through the use of barrier optimizations that incorporate implicit, explicit, and parasitic methods of information gathering..

2.1 Gang Scheduling

Gang scheduling [7] addresses the problem of time-sharing in a parallel environment by arranging the execution of parallel applications to provide the appearance that time-sharing is not occurring by guaranteeing that all processes are coscheduled. That is, it requires all processes participating in the communication to declare in advance their membership in a 'gang' that is explicitly coscheduled. This approach is generally very efficient for fine-grained applications where the processes that will be communicating can be easily be determined a priori, and communication takes place among a group of processes no larger than the number of processors available. Thus, gang scheduling requires modification in both the operating system and the application, and cannot readily adapt to changes in communication patterns. The particular implementation of gang scheduling described in [7] also requires hardware support (interrupt broadcasting) that may be unavailable for certain architectures (e.g., NOWs).

2.2 Demand-Based Coscheduling

Demand-based coscheduling [10] is conceptually similar to implicit coscheduling, in that it seeks to isolate application writers from programming model requirements and implicitly adapt to the communication patterns of the application. However, like gang scheduling, it assumes the ability to alter the operating system. Sobalvarro and Weihl simulate one demand-based coscheduling algorithm, dynamic coscheduling, for message-passing architectures and propose another, predictive coscheduling, for shared-memory machines with hardware-only enforcement of cache-coherency. In dynamic coscheduling, message arrivals sometimes cause preemption of the currently executing process in order to run the process for which the message was intended. In predictive coscheduling, the authors propose that interprocess communication can be efficiently detected and tracked through use of the virtual memory subsystem and the translation lookaside buffer (TLB) to determine the correspondents of each process. Furthermore, they contend that this historical information can be used to predict future communication with sufficient accuracy to improve performance if the scheduler attempts to coschedule those correspondents when switching to that process.

2.3 Other Coscheduling Strategies

In [2], Cosimo Anglano compares through simulation the efficacy of twelve different coscheduling strategies on a network of workstations. The strategies were obtained by crossing three different message-handling techniques (1. do nothing special when a message arrives; 2. boost the priority of the receiving process immediately; 3. periodically check the list of queued messages, and boost their priorities) with four different message-waiting techniques (1. spin waiting; 2. immediate blocking; 3. two-phase waiting [9]; 4. two-phase waiting that yields rather than blocks [8]). Anglano's primary result, not entirely unexpected, is that the best strategy depends on the expected workload of the NOW. Other results indicate that simpler strategies can often perform effectively, and that the particular scheduler in use by the operating system rarely effects the relative performance of a strategy.

3 Barrier Optimizations

3.1 Research Environment

The research environment consisted of Sun equipment, a mixture of 300 and 440 MHz Ultra 10 workstations running the Solaris 8 operating system, and a Netra cluster connected by Myrinet on which we had initially intended to use an implementation testbed. However, the differences between our environment and where the implementation originated proved insurmountable in the limited time available. As such, our investigations proceeded with the simulator used in [4].

The simulator implemented several different classes of algorithms, each configurable at the command-line. Four schedulers were available: the best-case explicit coscheduler; the worst-case time-sharing scheduler (TS), modeled after the one used in Solaris; and two fair stride schedulers, one with a system credit (SSC) extension that compensates processes who voluntarily give up their remaining quantum and one without (SS). We did not investigate scheduler tweaking. Initially, there were two kinds of barriers available, all-to-all, and all-to-one, of which the latter performed substantially better. The simulator also handled a number of waiting algorithms. The first, WAIT_BLOCK, simply blocks immediately after receiving a message, and performs poorly, about a third as well as WAIT_PAIRWISE, the implicit coscheduling algorithm. WAIT_SPIN_FIXED spins for a fixed amount of time after receiving a message, and performed about as well. Other algorithms provided by the simulator either performed exactly the same as another or were not relevant to our study.

We assumed the following test parameters to simulate the class of applications whose performance we wished to improve. For each test there are three jobs, each using the CPAT_CONT_NEWS communications pattern, in which every node continuously communicates with four of its neighbors. Each job communicates every 10us, and synchronizes every 150ms, with a 1.5 load imbalance factor. For other simulator parameters, the defaults were used as they seemed to represent fairly normal numbers for a cluster: a 10ms quantum, 10us network latency, 2us overhead, and a 100us context switch cost. The test configuration was simulated with three different job lengths: short, average, and long corresponding to 10, 50, and 500 barriers respectively.

3.2 Tree-Based Barrier Messaging

There are two common types of barrier synchronization: all-to-all and root-based. For all-to-all barriers, each process is responsible for notifying all coprocesses when it reaches the barrier. The barrier completes and computation ensues when every process has received notification from every coprocess. In root-based barriers, one process is designated as the root of barrier communication, and as such is responsible for sending barrier completion notifications to all coprocesses once it has received an arrival message from every participating process. Root-based barriers thus result in the association of much of the barrier message overhead to the root process. A common approach for handling the linear increase in overhead as the number of job coprocesses increases is the use of tree-based barrier messaging. In tree-based barrier messaging, arrival notifications and/or completion notifications are distributed among many coprocesses using tree structured directed graphs. Typically, binary or binomial trees are chosen as the structure for distributing notifications. The original simulator did not employ such barrier overhead optimizations, and the root process handled all barrier notification messages using a sequential flat tree. The decision was made that our study would evaluate the effects of allowing root-based tree-structured barrier messaging, in hopes of obtaining improved performance due to the overlapping of message processing and communication. We examined three types of tree structures for improving barrier performance. Each structure handled only barrier completion notifications, for reasons that will be described later.

3.2.1 Broadcast Tree

The first tree structure implemented was that of a broadcast tree, typically used for distribution of large messages. Broadcast trees are a type of binomial tree organized such that at each time step, every node that has received the data in a previous step is sending the data to a node that has yet to receive it. The organization for a broadcast tree for 16 nodes is shown in Figure 1; each edge is labeled with the time step upon which the edge is traversed. Notice that a broadcast tree completes in the same number of steps as would be used in a binary tree, that being $log_2(n)$, where n is the number of processes in the job. The results from running the base simulations with the new broadcast tree based barrier completion notifications were surprising, showing a slowdown over the base results for all three schedulers for average and long jobs. After further analysis, the observation was made that the initial step in the tree may be a source of slowdown. Broadcast trees take advantage of pipelining to amortize the cost of the first large transfer, which is dominated by transmission time as opposed to latency (network + protocol overhead). However, since the simulated barrier notification messages are each only one-byte long, the first step of sending from the root process to the first coprocess is dominated by latency. The minimum time for the message to reach the first coprocess is $L + 2o$ (where L is the network latency and o represents protocol overhead), but during that time the root process could send $(L + 2o)/o$ additional messages. Using this information, the second type of tree structure was designed.

3.2.2 Skewed Broadcast Tree

The second tree structure explored attempted to take advantage of the knowledge of the time needed for the first message to reach its destination by allowing the root process to continue sending messages during that period. The structure, referred to as the skewed broadcast tree, is shown in Figure 2 for 16 nodes. In this tree organization, each node acts as before the broadcast part of the tree does not begin until after the expected first message arrival. The results from using the skewed broadcast tree verified that this optimization was only slightly effective, as improvement over the original broadcast tree was achieved for only average and long jobs on two of the schedulers, time-sharing and stride. The results from this tree configuration were still disappointing, showing no consistent improvement over the base results. Once again, the situation required further analysis and another key observation was made. In neither the original or skewed broadcast trees was any thought given to whether or not the interior node processes contacted by the root process were actually scheduled. Instead, the algorithms used only the process numbers to determine interior nodes. As a result, some or all of these interior node processes may actually be asleep when the root first sends them a message, and all children of the sleeping nodes will not receive notifications for at least the time of a context switch. In this case, the original flat tree would still outperform the skewed broadcast tree, since the root process can send notifications to all coprocesses in a period much shorter than the length of a context switch. Accordingly, the idea for the third tree organization was formed in order to avoid sleeping interior nodes.

3.2.3 Order-Aware Skewed Broadcast Tree

The final tree structure examined attempted to use information on when processes registered for the barrier in order to predict which processes would be scheduled upon barrier completion. These processes could then be used as the interior nodes for the skewed broadcast tree in Figure 2, resulting in the order-aware skewed broadcast tree. In order to implement this tree structure, the root process must keep track of the order in which each coprocess reaches the barrier. The processes selected for use as interior nodes are then the last few to reach the barrier, based upon the assumption that these processes are most likely to still be scheduled. Unfortunately, due to a lack of time and an as yet unexplained bug in the simulator for longer jobs using the SS and SSC schedulers, the order-aware skewed broadcast tree could only be tested for the TS scheduler and short jobs. For short jobs, the order-aware version did improve upon the performance of the order-ignorant version for TS and SS. The results for longer TS jobs were less encouraging, once again proving no performance gains over the original flat tree. As was the case for the previous tree, the lack of improvement is suspected to be a direct result of the selection of sleeping interior nodes. Further investigation is warranted toward finding a better predictor of processes that will be scheduled upon barrier completion, in hopes of realizing the benefits of the order-aware skewed broadcast tree.

3.2.4 Other Methods for Distributing Barrier Overhead

As noted before, the above structures were used only for testing improvements gained in their use in barrier completion notifications. Since no conclusive evidence was found that more complex tree organizations could improve performance of completion notifications, optimization of registration notifications was not attempted. Such optimizations should not be ruled out for future study, however, since registration trees may be less likely to suffer from the sleeping interior node problem, but this remains to be seen. In addition, registration trees may serve useful as a means to gather parasitic information that could be used to influence non-root process waiting behavior or help determine which nodes will be scheduled at barrier completion.

One might also ask why the simpler binary tree structure was not evaluated. The reason for choosing against the binary tree is that it is just as (or even more) susceptible to the problem of sleeping interior nodes, and does not take full advantage of using a node that is scheduled to do more than two notifications.

3.3 Alternative Waiting Strategies

Our analysis of waiting algorithms began by considering in the abstract why it might be that applications with infrequent barriers and continuous communication behave so poorly in the context of large load imbalances. Our intuition into the fundamental difficulty is that the delay between message arrival and message response can be attributed to the sleep induced (and therefore, context switch required) by the lack of messages resulting from the load imbalance during the base synchronization spin. According to this hypothesis, spinning for the maximum load imbalance plus the message latency (V+L+2o) ought to prevent loss of synchronization, and thereby improve performance. As such, this behavior was implemented as a variation of WAIT_PAIRWISE with a large initial spin. To our consternation, however, the simulation results (in Table 1) showed that this was not the case. After reconsideration, several conclusions were put forth: (1) given that the load imbalance in the job was much greater than a single quantum, odds were good that a process would be pre-empted and slept if it tried to spin for the entire V+L+2o; (2) since Split-C implements data transfers as pulls, rather than pushes, messages may arrive to a process after it has reached the barrier; and (3) that therefore, the primary inefficiency at barriers in the presence of load imbalance stems from the increase in response time due to context switches on the part of the processes that have reached the barrier and subsequently blocked.

We realized that more information would be needed in deciding the appropriate action to take at a barrier. Initially, one such piece of information that looked promising was the knowledge of the processes currently scheduled. Each coprocess would collect this information implicitly (by the normal implicit co-scheduling mechanism of inference), and parasitically sent to the root process with barrier registrations. When a process joins the barrier, the root would collate this parasitic information, and return the state, insofar as it can be inferred, of the system. The process can then try to use this information, and its knowledge of which processes needed data from it either most recently or in the most recent barrier, to predict when it should be awake. However, due to time constraints, we were unable to implement an algorithm which collected and used this information. Furthermore, making a local decision about when to wake up implies the ability to sleep until a specific time, or for a specific amount of time, an ability which may not be generally available.

Our efforts then turned to the use of information that would be easier to extract from the simulator, and had plausible implementation possibilities. Experimentally, it was desirable to see if foreknowledge of the imbalance of the job would be useful. The first algorithm implemented, VARIANCE_GUESS, thus used knowledge of the variance to predict when a barrier should complete. If this time was more than a pair of context switches away, the process blocked. This algorithm, while not horrible, did not result in a performance increase. Next, our attention returned to the idea of collating implicit data transported parasitically and distributed explicitly, but in this case, the data would be the number of processors to have joined the barrier thus far. This information could be used by the second algorithm implemented, VARIANCE_WITH_PROC_COUNT, to better tune its predictions by implicitly accounting for unscheduled jobs. The additional information proved valuable, as the performance of the first algorithm was improved by enough to compete with conventional WAIT_PAIRWISE waiting. Entirely removing (fore)knowledge of the imbalance and predicting the expected barrier completion time by measuring the current elapsed time divided by the number of processes to have joined the barrier, the AVG_PROC_DELAY algorithm, performed similarly to VARIANCE_GUESS. As would be expected, the more information available in making local decisions, the better the performance.

At this point, the decision was made to take a closer look at WAIT_PAIRWISE waiting, in an effort to answer the following questions. What would happen (1) if WAIT_PAIRWISE blocked immediately when it hit a barrier (BLOCK_IMMEDIATELY)? (2) if it tried to spin forever at a barrier (SPIN_FOREVER)? (3) if it spun for a varied amounts of time initially (LARGE_INITAL_SP
600, 750, 22500, 25000, 27500})? When WAIT_PAIRWISE blocks immediately, it slows down by roughly a factor of three; if it tries to spin forever, it slows down by a factor up to one hundred. Clearly, it is important to block when not performing useful communication or computation, in order to allow other processes a chance to do useful work. On the other hand, it is also important not to block immediately, and repeatedly pay the cost of a context switch. While a few of the initial spins improved on WAIT_PAIRWISE's default initial spin (3 times the context switch, or 300us), none of them did so consistently, and the values were not generated algorithmically. It is inconclusive as to why certain values perform worse or better than others, as no relationship to the simulation parameters could be determined.

None of the gains from altering the initial spin were particularly impressive, so we began to think that we made a mistake by concentrating on the initial spin. Given our hypothesis that the major cost was the increase in response time caused by context switches, the obvious step was to try to reduce this cost. One approach, which has not been attempted due to time constraints, is to have each process attempt to predict the processes with which it will be communicating at time $W+L+2o$ from now, and send them a message to either cause a blocked process to be scheduled or keep a scheduled process spin waiting. As a result, those processes would be awake when the read arrived, and the read would complete quickly. This approach is similar to using hardware prefetching to hide memory latency. Another approach is to try to probabilistically improve the odds of a process being scheduled when it should be, that is, whenever another process attempts to communicate with the process. In effect, this is what increasing the initial spin time attempted to achieve. However, if reads are not clustered at the beginning of the barrier, such an approach will be ineffective. To improve the odds of being scheduled when contacted, the spinning should be better distributed (evenly throughout the barrier, clumped at the end, or clumped in the middle) according to however the reads are distributed, but this information is application dependent. There are several straightforward ways to distribute the spin time throughout the barrier. One approach is to send an empty wake-up message to every process every time a process registers with the root process for the barrier. A second would be to send wake-up messages at some periodic interval, similar in action to a heartbeat protocol. The next section details our attempts into gaining performance increases through the use of the former method.

3.4 Explicit Keep-Alive Synchronization

As described in the introduction, implicit coscheduling only performs unacceptably for applications where there is high communication and load imbalance and infrequent synchronization. For applications with no load imbalance, processes generally progress at the same relative speed. As such, barrier synchronization takes advantage of the conditional two-phase waiting since processes often exchange data immediately prior to the barrier. One hypothesis that may be formed to explain the poor behavior in the presence of load imbalance is that as processes reach the barrier, there is no longer sufficient communication occurring to keep them scheduled. Although this seems unlikely for continuous-communication applications, possible explanations for this phenomena are that the processes with whom they typically exchange data are either presently doing computation or have themselves blocked due to a load-imbalance induced lack of coscheduling. In order to test the above hypothesis, an investigation began to evaluate the use of explicit keep-alive messages sent from the root node to all other coprocesses during barrier operations in an effort to prevent processes from sleeping due to a lack of communication and occasionally wake other processes that have blocked to improve coordination.

In devising such a keep-alive strategy, there are two important factors that one must take into account. The first factor is the amount of communication that must be received by a process using two-phase waiting in order to remain spinning. The current implicit coscheduling algorithm (WAIT_PAIRWISE) in the simulator uses a function of the specified communication spin and synchronization spin to determine this value, which is approximately one to two messages for the default parameters. As a result, the root process need not send a large number of messages to keep coprocesses spinning. The second consideration in developing an appropriate strategy is the relative time to complete a barrier operation for applications with infrequent synchronization and high load imbalance. Experimental results from the simulator reveal that the average barrier operation takes anywhere from 10 to 70 times the length of a scheduler quantum for such applications. Using this information, it is advisable that processes that reach the barrier early not spin for their entire quantum every time they are scheduled until the barrier completes, as this takes away valuable computation time from other jobs executing on the system.

Based upon the previous observations, the algorithm for performing keep-alive synchronization was structured as follows. Each time a barrier registration message arrives at the root process, a single one-byte message is sent to all coprocesses. The result of this message on coprocesses depends on the state of the process when the message is sent. If a process is already spin waiting, the message will serve to extend the waiting period of that process by at least one conditional spin time, according to the conditional two-phase waiting algorithm. If a process is blocked, the message will trigger the scheduling of the process in the near future. The combined result is that most of the processes will be scheduled at the same time, allowing processes that have not reached the barrier to communicate with their coprocesses and make further progress. In order to test the hypothesis that processes that have reached the barrier early should not spin but rather allow other jobs to use the time for computation, the time at which the root began to send keep-alive messages was varied based upon the fraction of coprocesses to have reached the barrier. The performance results for various keep-alive levels versus the baseline performance (orig) are shown in Figure 3 for all three scheduling disciplines and application lengths.

The results in Figure 3 are very encouraging, showing positive performance improvements for all but short running TS jobs. For each of the various keep-alive starting levels and schedulers, the performance benefits seem to converge upon the values for long running jobs, approximately 4-6% speedup in total execution time for the best performing level for each scheduler. Both the TS and SSC schedulers show best performance when the root process only initiates keep-alive messages after $\frac{3}{4}$ of the jobs coprocesses have reached the barrier. The SS scheduler benefits from an even later starting level, where $\frac{9}{10}$ of the processes have registered for the barrier. The difference in the best performing starting levels can be attributed to the fact that the TS and SSC schedulers both compensate processes for the time they were sleeping, by increasing the priority in the case of TS and through the granting of exhaustible tickets under SSC. As such, processes under these two scheduling disciplines are more likely to be scheduled immediately upon keep-alive message reception in order to compensate for the time they have been asleep, so sending messages earlier has a greater chance of having the desired effect.

Although different schedulers perform better for different starting levels, the results do confirm the hypothesis that starting keep-alives later in the barrier does produce the best performance, as performance for the $\frac{1}{4}$ and $\frac{1}{2}$ levels are worse than those for later starting levels in the case of long running jobs. As further proof, we decided to analyze the percentages of time spent in the various job states (computation, communication, synchronization, idle, and swapping) for the original and best case performance. Figure 4 shows the relative percentages for each job state for all three schedulers and job lengths. In each of the three graphs, it is easy to see that the performance increases were actually gained largely in reducing idle time, with slight gains from reducing swapping. So does this mean that our hypothesis is incorrect? One might think that the keep-alives could just be changing process idle time into synchronization spinning time. If this were in fact the only effect, we would see an increase in synchronization time equal to the decrease in idle, and the overall time would not vary. As such, the keep-alive mechanism must be decreasing both idle and synchronization time, corresponding to the notion that idle time is being converted to synchronization time, which itself is being reduced due to the additional coscheduling gained by explicitly contacting coprocesses near the end of the barrier to ensure quick progress toward barrier completion. This explanation also illuminates the slight reduction in swapping, since a keep-alive will serve to prevent processes that are spin waiting when the message arrives from blocking due to a lack of communication.

4 Conclusions & Future Work

We conclude that implicit coscheduling is difficult to improve upon since compensating for the irregularities inherent in applications with large load-imbalance and infrequent barriers would require a great deal of information. Further work may be possible in using this information to predict when to a process should be awake, but will suffer from the overhead necessary to acquire information sufficient for prediction, even assuming the ability of a sleeping process to influence its wake-up time. We found that more complicated structures for barriers were ineffective because of scheduling irregularities, and, due to the short messages involved, are unsure if the potential benefit could outweigh the overhead necessary to compensate.

Our only improvement over the default barrier strategies employed by implicit coscheduling came from attempting to probabilistically compensate for irregularities by (more or less arbitrarily) waking up sleeping processes so they could service reads from processes which had not yet joined the barrier. Many improvements on our initial implementation are possible, especially in tuning which processes receive wake-up messages, as non-joined processes are assumed to be doing useful work. Variations on our implementation of explicit keep-alives – periodic keep-alives, a combination of periodic and arrival-based keep-alives, and different thresholds, e.g., elapsed time in the barrier – should be investigated.

Finally, we believe that 'prefetching' – forcing a remote process which will be read from soon to be swapped in – could offer significant benefits. Not only could it eliminate what we identified as the major cost (in proportion to its predictive accuracy), but it differs from the other approaches in potentially important ways. Rather than simply reacting to local condition (WAIT_PAIRWISE), or attempting to probabilistically compensate for global conditions (explicit keep-alives), or to predict remote actions (when should I be awake so other processes can read from me?), it attempts to analytically predict local actions, and decide when remote processes should be awake so it can read from them. Analytical predictions should be more accurate, enabling larger improvements, and since the analysis and prediction remains local, overhead should be negligible. Furthermore, the use of gray-box techniques [6] applied to the running parallel job(s) as well as to the operating system on which it runs may prove useful in collecting such analysis.

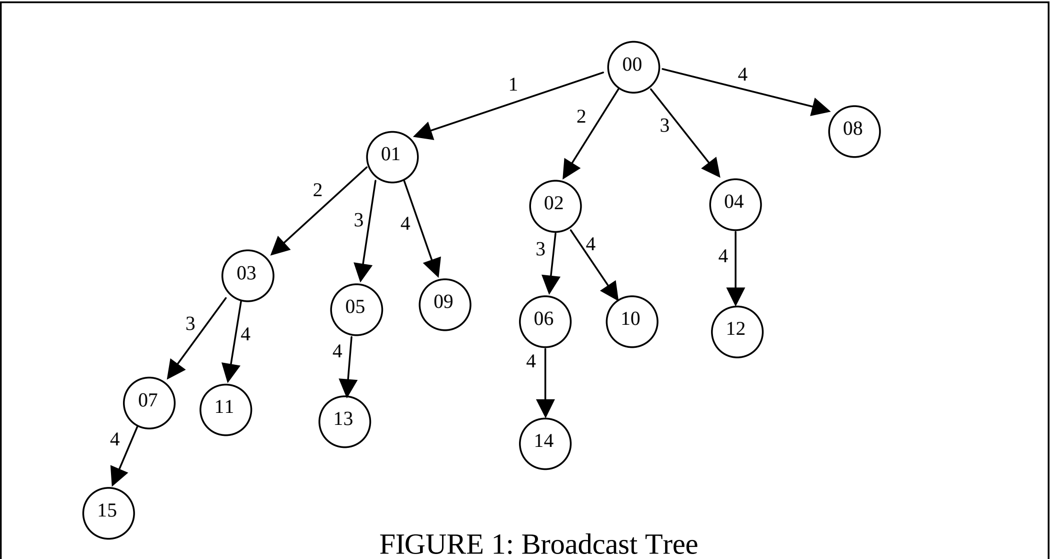


FIGURE 1: Broadcast Tree

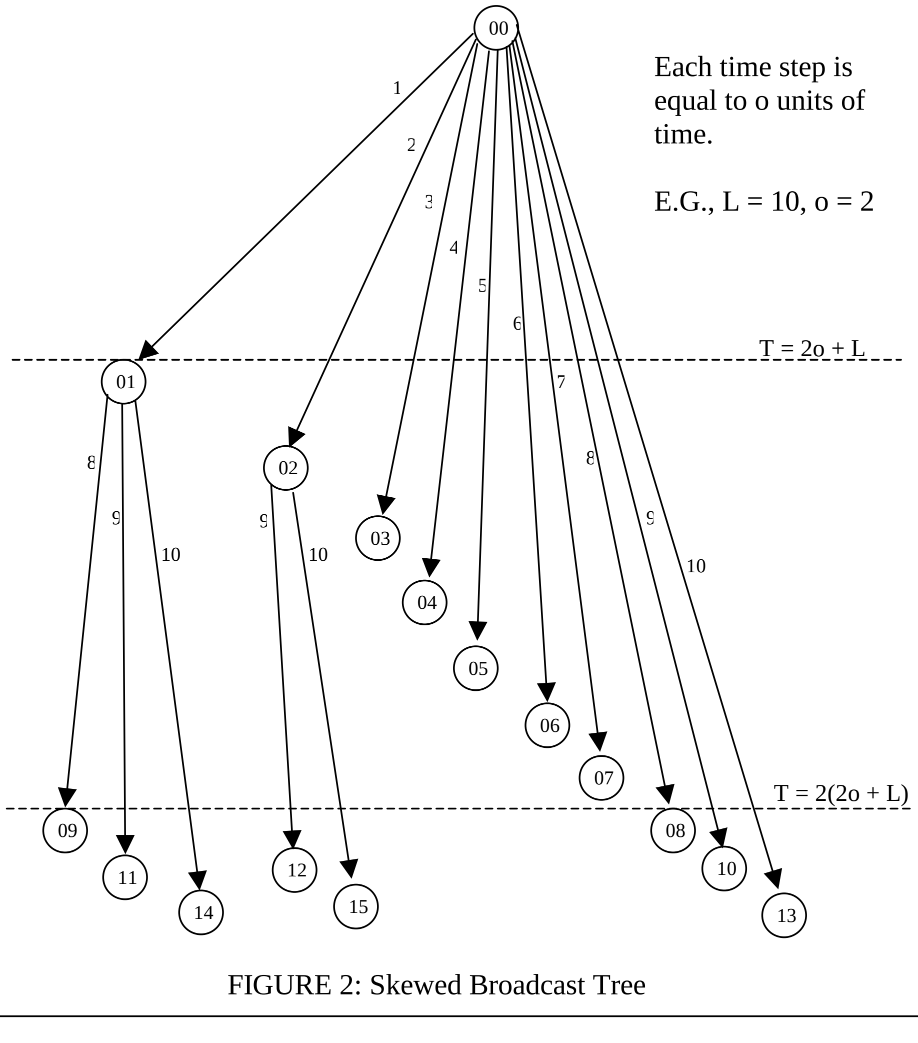
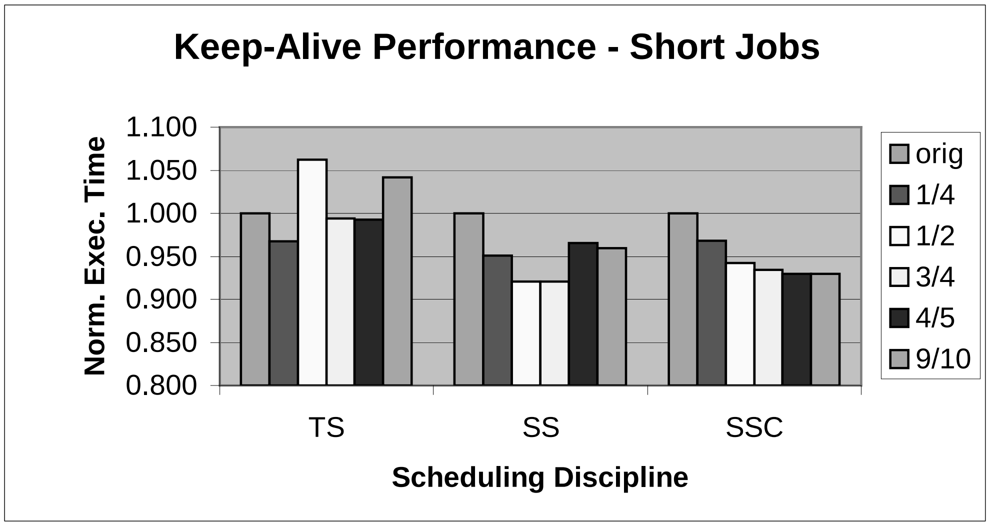
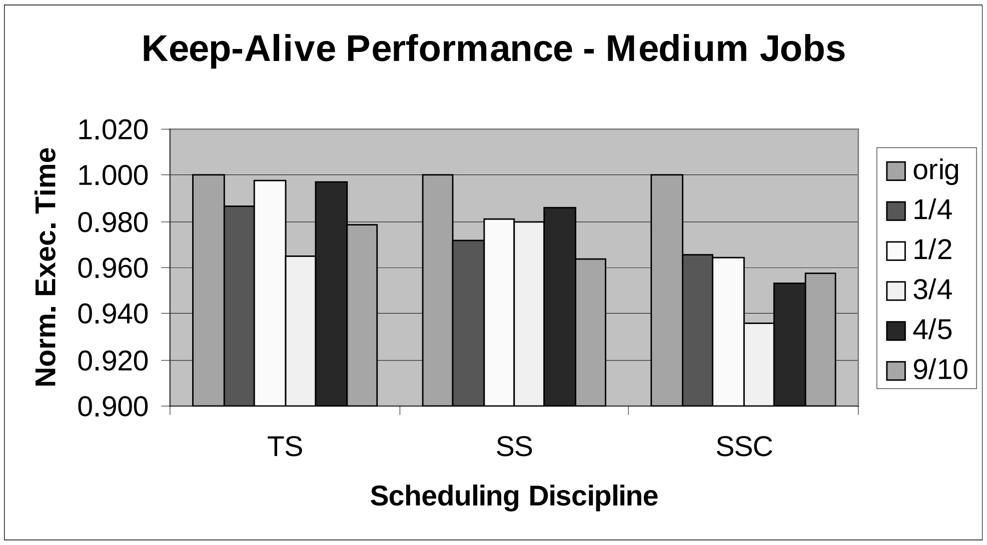


FIGURE 2: Skewed Broadcast Tree

Table 1: Performance of Alternative Waiting Algorithms

Algorithm	Scheduler	Short	Medium	Long
AVG_PROC_DELAY	SS	6.43	38.33	387.90
	SSC	7.00	39.33	401.94
BLOCK_IMMEDIATELY	SS	20.69	121.74	1255.53
	SSC	19.70	115.36	1209.09
LARGE_INITIAL_WAIT	SS	9.48	51.03	520.95
	SSC	10.76	59.40	622.96
	TS	10.97	67.13	687.44
22500	SS	6.40	37.93	392.42
	SSC	6.43	38.6	404.38
	TS	6.89	42.52	438.97
25000	SS	6.40	37.26	407.64
	SSC	6.68	39.73	389.98
	TS	7.28	42.51	443.00
27500	SS	7.07	39.28	309.38
	SSC	6.79	39.40	386.91
	TS	7.33	42.53	423.59
450	SS	*5.98*	35.83	379.06
	SSC	6.38	37.64	*386.91*
	TS	7.20	40.33	*423.59*
600	SS	*6.04*	37.13	382.74
	SSC	*6.22*	38.05	392.73
	TS	6.72	40.85	*423.69*
750	SS	6.32	36.84	379.65
	SSC	6.89	38.06	*388.54*
	TS	6.69	41.18	*422.67*
SPIN_FOREVER	SS	214.92	1326.57	[error]
	SSC	1519.18	[error]	[error]
VARIANCE_GUESS	SS	6.43	38.80	392.38
	SSC	6.67	39.92	407.90
	TS	7.07	41.56	430.31
VARIANCE_WITH_PROC_COUNT	SS	6.22	36.94	382.22
	SSC	6.47	38.50	391.50
WAIT_PAIRWISE	SS	6.15	35.72	378.91
	SSC	6.27	37.53	391.15
	TS	6.72	39.94	424.36





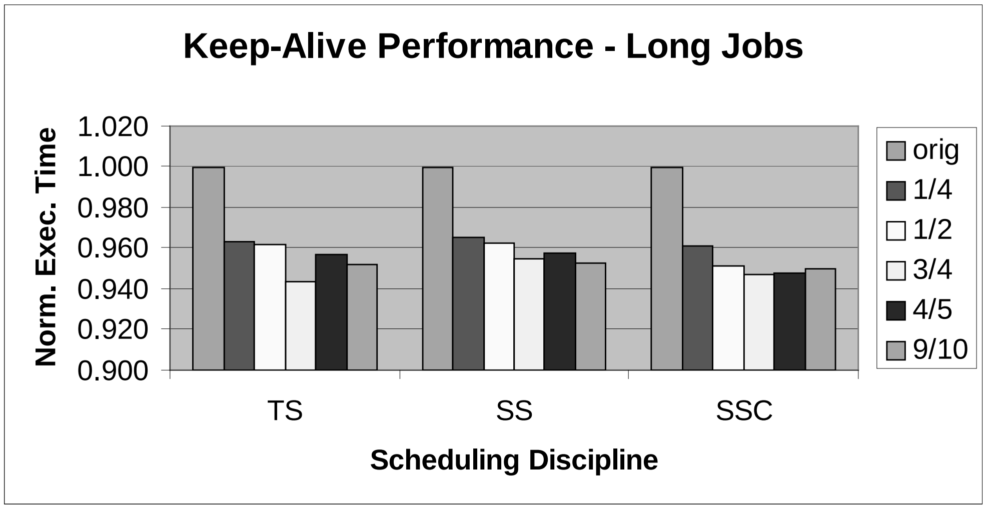


FIGURE 3: Keep-Alive Performance

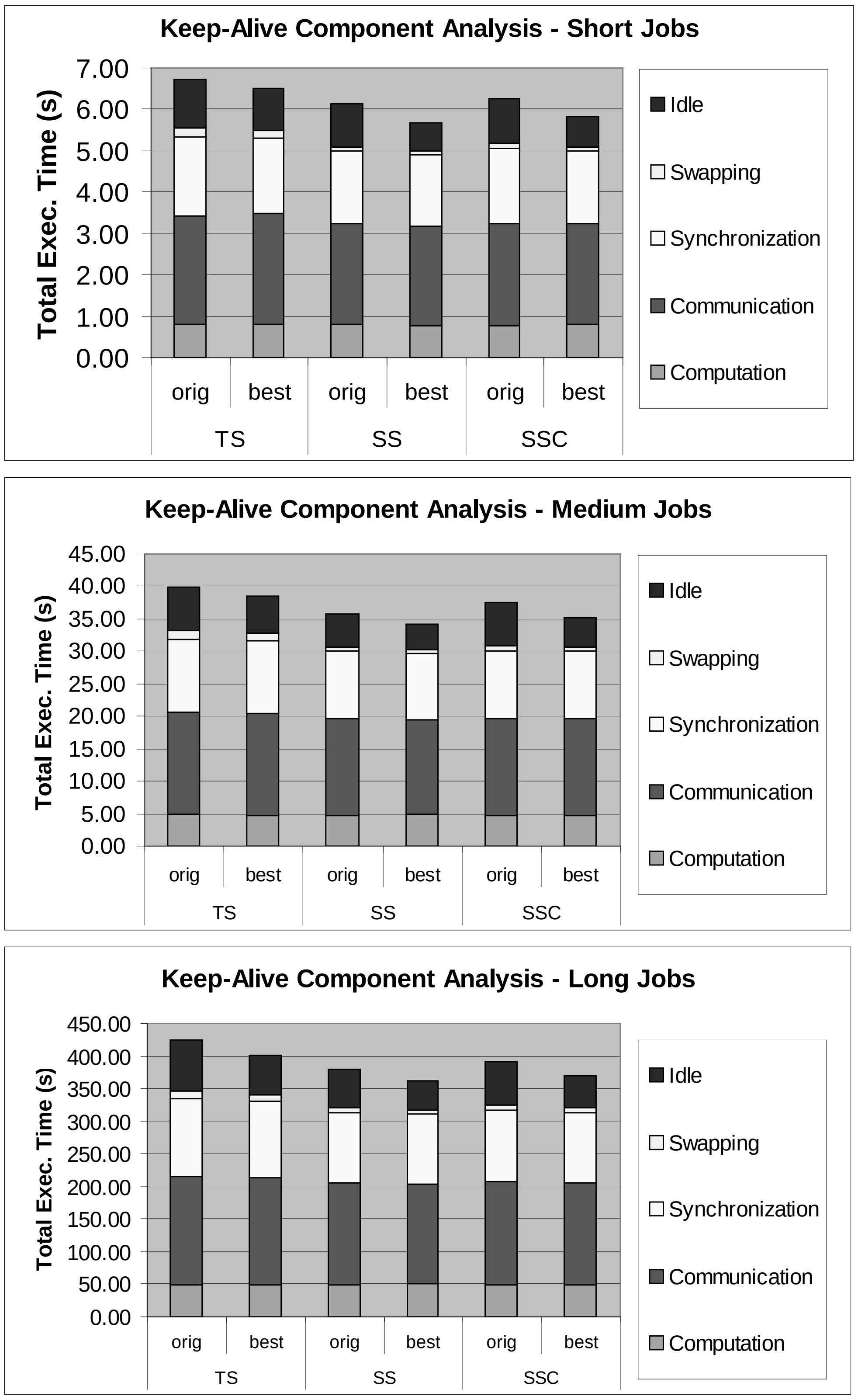


FIGURE 4: Job State Analysis

References

[1] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations, 1995.

[2] C. Anglano. A comparative evaluation of implicit coscheduling strategies for networks of workstations, 2000.

[3] Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. An information-based approach to distributed systems design, 2000.

[4] Andrea C. Arpaci-Dusseau. Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. Technical Report CSD-99-1052, University of California, Berkeley, 10, 1999.

[5] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick. Parallel programming in split-c, 1993.

[6] Andrea C. Arpaci Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-box Systems. In *Proc. of The Eighteenth Symposium on Operating Systems Principles (SOSP '18)*, October 2001.

[7] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[8] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. Alternatives to coscheduling a network of workstations. *Journal of Parallel and Distributed Computing*, 59(2):302–327, 1999.

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

[10] Patrick G. Sobalvarro and William 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.

[11] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation, 1992.

[12] M. Warren. Parallel supercomputing with commodity components, 1997.