Scheduling Techniques for Concurrent Systems

John K. Ousterhout

Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Current operating systems base many of their decisions on the assumption that processes are independent. This paper examines what happens in multiprocessor systems that use interprocess communication extensively. Traditional techniques for short-term scheduling result in serious limitations on interprocessor communication; a two-phase blocking scheme is suggested as a solution to the problem. Long-term scheduling policies that assume process independence result in a form of thrashing when there are groups of cooperating processes. The notion of *coscheduling* is introduced, and three algorithms are described for achieving it. Simulation results suggest that substantial degrees of coscheduling can be achieved over a variety of conditions using relatively simple methods.

1. Introduction

The scheduling techniques used by current operating systems are based in large part on the assumption that processes are independent. It has been assumed that interactions between processes are the exception rather than the rule and, until recently, this has been the case. In more recent systems, however, the assumption of process independence is becoming less and less valid. Multiprocessor systems are appearing more and more frequently; they encourage a style of programming wherein collections of cooperating processes use several processors concurrently to solve problems. Cooperating groups of processes are being used even in single-processor systems; perhaps the most visible example of this approach is the pipeline mechanism of Unix [Ritchie 74].

The work described here was performed at Carnegie-Mellon University. It was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551, and in part by the Graduate Fellowship Program of the National Science Foundation.

As cooperation between processes becomes more widespread and occurs on a finer grain, traditional scheduling techniques break down. This paper describes two ways that naive schedulers can limit the efficiency of interprocess communication and presents simple and practical mechanisms to eliminate those bottlenecks. The mechanisms described here were implemented in the Medusa operating system [Ousterhout 80].

Section 2 shows how short-term scheduling can become the bottleneck in interprocess communication, and suggests a two-phase blocking mechanism as a solution. The remaining sections of the paper address a more difficult problem associated with long-term scheduling. Section 3 shows how a form of *process thrashing* can occur when traditional mechanisms are used to schedule cooperating processes, and introduces the notion of *coscheduling*. Section 4 describes three coscheduling algorithms. Section 5 presents simulation results that illustrate the differences between the algorithms. Although the algorithms are relatively simple and were designed for a more restricted environment than exists on many multiprocessors, the simulation results suggest that all three will provide acceptable degrees of coscheduling.

There are strong similarities between the problems of coscheduling and the problems faced in memory management. The paper introduces analogs for thrashing and working sets and shows how the various coscheduling algorithms make tradeoffs between forms of external and internal fragmentation. A technique similar to bit-map memory allocation appears as part of one of the coscheduling algorithms.

2. Making Waiting Efficient: Pauses

Waiting is a fundamental aspect of communication. When information is to be passed from one process to another, it is exceedingly unlikely that the two processes will both reach the *rendezvous point* at exactly the same instant in time. Either the sender will have the information ready before the receiver is prepared to accept it or vice versa. Thus one of three forms of waiting must occur: either 1) the receiver will wait for the sender to reach the rendezvous point (because no data is available to be received), 2) the sender will wait for the receiver to reach the rendezvous point (because the communication mechanism is unbuffered or the available buffer space is full), or 3) the data will wait (in a buffer) for the receiver to reach the rendezvous point.

In situations where information flows in only a single direction, for example from a program to a disk file, clever buffering can be used to ensure that the slower

process (the bottleneck in overall throughput) never has to wait. Under these conditions, case 3 dominates; current operating systems make that case quite efficient. However, as interprocess communication becomes more and more prevalent, two-way (interactive) information flow is becoming more common. Perhaps the best examples of two-way flow are the remote procedure call mechanisms being implemented at several sites [Nelson 81, Spector 82]. In remote procedure call, a *client* process sends a message to a *server* process and immediately waits for a reply. The server process waits for a request from a client process, serves it, and sends a reply. In the Medusa system, operating system functions are invoked with a form of remote procedure call. Remote procedure call and other forms of two-way communication lead naturally to process waiting (cases 1 and 2 above) rather than data waiting (case 3 above).

Most short-term schedulers make process waiting very inefficient. Whenever a process waits for some event, it is thrown off its processor and another process is activated. When the event occurs, the process must be reactivated. In single-processor systems there is no way to avoid the two context swaps since another process must execute to generate the event. However, in a multiprocessor system the communicating processes generally execute on different processors, so the context swaps may waste time unnecessarily.

A solution is to divide waiting into two phases. During the first portion of the wait, called a *pause*, the execution state of the process remains loaded and the processor is idle. If the desired event occurs during the pause time, the process can be reactivated with no context swapping overhead. If the pause time is exceeded, then the process enters the *block* phase and must relinquish its processor. Pauses are designed especially for the situation where communication is occurring on a very fine grain, i.e. the event being waited for is likely to occur very soon after the wait begins. If the duration of the wait is less than two context swap times then the lost processor time due to the pause is less than the time that would otherwise have been wasted in context swaps. If waits generally last a long time, then the pause time should be made zero in order to avoid wasted processor time (however, in this case pauses occur infrequently so the total wasted time is likely to be small anyway).

The Medusa system implements pauses with a user-settable pause time. Although a context swap takes about as long as 150 average instructions, a message can be transmitted from one process to another process (paused on a different processor) in the same time as about 60 average instructions. The time given is the time from intiation of the send operation to the execution of the first instruction by the paused process after it receives the message and resumes processing. Of this time, only about 10 instruction-times are used to reactivate the process; the rest is used to transmit the message. Thus a paused receiver will begin servicing a message 3-4 times as fast as a non-paused receiver (60 instruction times versus 210).

3. Thrashing and Process Working Sets

Long term scheduling policies can also limit the speed of interprocess communication. This is most likely when communication occurs much more frequently than the scheduling decisions that establish priorities. For example, suppose several processes are executing on multiprocessor and sending and receiving messages among themselves. Now suppose that the system's scheduling policy allows half of the processes to execute only in odd time slices and the other half only in even time slices. If the processes are interacting frequently, then it is likely that most or all of the processes in the executing half will block awaiting messages from processes in the descheduled half. When the descheduled half is activated and the (blocked) half is descheduled, then the newly-executing processes are also likely to block while waiting for messages with the idle half. Regardless of the raw speed of the processes or of the interprocess communication mechanism in this example, the processes will only be able to interact as frequently as the system reschedules processors. Even more intelligent schedulers than the one in this example can produce similar behavior unless they take account of the communication patterns of the processes.

There is a strong similarity between the above example and the thrashing that occurred in early demand paging systems. Just as there is a working set of memory pages that must be coresident for a uniprocessor program to make any progress (for example, see [Denning 70]), closely-interacting parallel programs have a *process working set* that must be *coscheduled* (scheduled for execution simultaneously) for the parallel program to make progress. In memory thrashing, the swapping-in of each page needed by the program causes one or more of the other pages needed by the program to be swapped out; thus the progress of the program is limited by the speed of swapping, not the speed of references to main memory. Similarly *process thrashing* occurs when the scheduling of each process whose services are awaited causes another process, whose services will soon be needed, to be descheduled; the progress of the parallel program is limited by the rate at which scheduling decisions are made, not the speed of the low-level communication primitives. If efficient interprocess communication primitives are to be used to their fullest, mechanisms must be provided to avoid process thrashing.

Two steps must be taken to eliminate process thrashing: first, there must be a way of determining process working sets; second, the process allocation and scheduling mechanisms must be organized in a way that coschedules these working sets. Coscheduling, combined with the pause mechanism described in Section 2, allows the communication mechanism to proceed at full speed by ensuring that processes are available for interactions when needed. Coscheduling also reduces the number of context swaps arising from blocked processes and thereby reduces operating system overhead.

In demand paging systems the working set estimates are made dynamically. Hardware keeps track of the memory pages most recently touched by the processor. Since only one program runs at a time the access information can be used to estimate the working set for the current program; the operating system then ensures that the working set is main-memory-resident whenever the program runs. Any page may reside in any main-memory frame, hence the working set may be adjusted dynamically without massive shuffling of pages.

Unfortunately, several complicating factors make dynamic estimation of process working sets rather difficult. On a multiprocessor there may be many processes from different working sets executing at the same time; the mere fact that two processes execute a communication primitive at the same time does not imply that they should be in the same working set. One way to determine process working sets is to record information in the form of pairs of processes that

interact; both gathering this information and analyzing it to estimate working sets appear to be expensive operations.

A second problem with dynamic process working set estimation stems from *processor preferences*. In some multiprocessor systems a process can execute equally well on any processor (this is analogous to demand paging systems where any virtual page may be loaded into any physical page); on other multiprocessor systems with non-uniform memory a process must be run on the single processor containing its code and/or data (this is equivalent to memory management without mapping: a virtual page can reside in only one physical page). If processes have processor preferences then working set information must be available at the time the processes are assigned to processors.

3.1. Assumptions and Goals

Because of the difficulty of collecting and using process working set information dynamically, the rest of this paper considers a restricted scenario in which process working sets are specified statically by the programmer. Each program is a *task force* containing one or more processes that constitute a process working set. Each process belongs to exactly one task force, and it is assumed that the membership of a task force does not change during its lifetime. I assume a special form of processor preference that exists on Cm* [Swan 77]: a process can be loaded onto any processor, but cannot move after it starts execution. The ability to move processes dynamically would lead to better results than those reported here.

A task force is *coscheduled* if all of its runnable processes are executing simultaneously on different processors. Each of the processes in that task force is also said to be coscheduled. If at least one process of a task force is executing but the task force is not coscheduled then the task force is said to be *fragmented*; the collection of executing processes is referred to as the *executing fragment* of the task force.

The rest of the paper assumes that coscheduling is desirable, and examines how it might be achieved. The primary goal of the algorithms described here is to maximize coscheduling: for a given series of task force arrivals into the system and departures from the system, which cannot be predicted in advance, they attempt to maximize the average number of processors executing coscheduled processes. The coscheduling algorithms include both allocation and scheduling mechanisms. It is important that scheduling be efficient because it occurs frequently. Allocation occurs less often, so it can involve more complex computations.

The system is assumed to contain P processors, each of which can multiplex between at most Q processes (the simulation data uses P=50 and Q=16, which is the situation in Medusa). Thus process space consists of P*Q *process slots* to which processes may be assigned. Task force allocation consists of assigning each process of the new task force to an empty process slot. I assume that Q is large enough so that allocation always succeeds. Scheduling consists of ordering the execution priorities of the processes on each processor. The slot assignment of a process is not changed during its lifetime, and each process is required to execute on the processor to which it was initially assigned. The algorithms assume that no task force contains more than P processes.

4. Three Algorithms for Coscheduling

The three coscheduling algorithms discussed in this section derive their "flavor" largely from their views of process space. The first algorithm views process space as a two-dimensional matrix of processes, while the second and third algorithms view process space as a linear sequence.

4.1. The Matrix Method

The matrix coscheduling algorithm is a very simple one that performs surprisingly well. The space of all process slots is organized as a *matrix* with Q rows and P columns (see Figure 1). Each column contains the process slots of one processor, and each row of the matrix contains a process from each processor.

Allocation:
See if there are enough unused slots in row 0 of the matrix to accomodate all of the processes of the task force. If not, then attempt to assign all of the processes to slots in row 1, and so on until a single row is found that can hold the entire task force.

Scheduling:
This algorithm uses a round-robin mechanism to multiplex the system between the different rows of the matrix. In time slice 0, each process in row 0 is given highest execution priority on its processor, thereby coscheduling all task forces in that row. In time slice 1, row 1 is coscheduled, and so on until all task forces have been scheduled. Then return to row 0 and repeat.

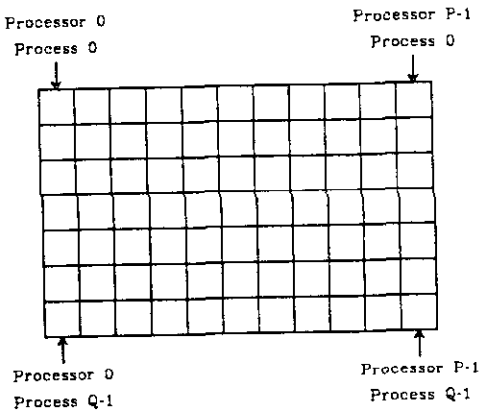


Figure 1. For the matrix algorithm, process space is organized 2-dimensionally, with each column containing the processes of a single processor.

4.1.1. Alternate Selection

When a row is scheduled for execution, it is likely that one or more of the process slots in that row will either be empty or contain processes that are blocked (e.g. while awaiting terminal input). When this happens, the processor scans its column of the matrix circularly for a process in another slot that is runnable. If a process in one of these slots is runnable, it is allowed to execute and is referred to as an *alternate*. Unless empty process slots cause all of the other processes in the alternate's task force to run, the alternate will execute as a fragment.

This alternate selection method is applied independently by each processor; no attempt is made to select alternates in a way that maximizes coscheduling, except that selection algorithm is identical in each processor. Thus it is likely that alternates will execute as frag-

ments; an alternate that really needs coscheduling will block as soon as it attempts interactions with other processes in its task force. However, the method does have the nice property that there is a clean separation between global and local scheduling decisions. At a global level, all that needs to be done is to select the high-priority row of the matrix. Given the number of that row, each processor can schedule itself and perform alternate selection independently. The simulation results presented below suggest that the increase in coscheduling to be had by selecting alternates centrally would be small; furthermore, a central scheduler would have to be invoked anytime there is a change in execution status of any process. In a large system, this might impose a considerable burden on the central scheduler. With the simple-minded alternate selection, changes in execution status can be handled locally by the kernels of the individual processors. Global intervention occurs only on time slice boundaries.

4.1.2. Summary of the Matrix Method

The attractive features of the matrix algorithm are the simplicity of its allocation and scheduling algorithms, and the clean separation between local and global scheduling decisions. The algorithm has several drawbacks. First, process space is partitioned into disjoint rows: a task force cannot be assigned to slots in more than one row. Thus there are likely to be many unused process slots in each row. This inefficiency is similar to *internal fragmentation* in paging systems. The other two algorithms are attempts to solve this problem; it turns out that they in turn suffer from an effect similar to *external fragmentation* in segmentation systems. A second disadvantage of the matrix method is the simple-mindedness of the alternate selection algorithm, which could cause opportunities for coscheduling to be missed. This disadvantage is shared by the other algorithms as well. Finally, the matrix algorithm causes the execution priority of the whole system to change at the same time. Thus shared facilities used for context swapping (e.g. paging devices) will be loaded unevenly.

4.2. The Continuous Algorithm

Most of the drawbacks of the matrix algorithm arise because of the rigid partitioning of process space into the rows of the matrix. The continuous algorithm uses a different view of process space in order to achieve denser packing and smoother scheduling (see Figure 2). For this algorithm, process space is viewed as a *sequence* of process slots. The process slots in any P consecutive positions of the sequence belong to different processors. The allocation and scheduling algorithms consider at a particular moment a *window* of P consecutive positions in the sequence, and slide the window across the sequence over time. For purposes of the algorithms below, process slot 0 in processor 0 is considered to be in the leftmost position of the sequence, and process slot Q-1 of processor P-1 is considered to be in the rightmost position.

Allocation:

Place a window of width P slots at the left end of the process sequence. See if there are enough empty slots in the window to accomodate the new task force. If not then move the window one or more positions to the right, until the leftmost process slot in the window is empty but the slot just outside the window to the left is full. Repeat this until a window position is found that can contain the entire task force.

Scheduling:

Place a scheduling window of width P slots at the left end of the process sequence. At the beginning of each time slice move the window one or more slots to the right until the leftmost process in the window is the leftmost process of a task force that has not yet been coscheduled in the current sweep. When the window has advanced far enough that all existing processes have received execution time, return the window to the left side and start a new sweep. When the window contains empty process slots or processes that are not runnable, use the alternate selection mechanism from the matrix algorithm.

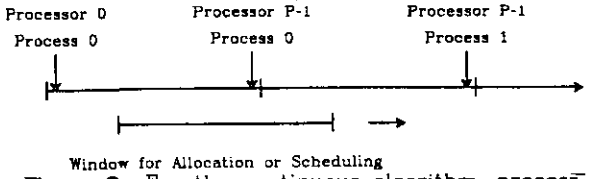


Figure 2. For the continuous algorithm, process space is organized as a linear sequence of process slots; at any given time the scheduler or allocator considers those slots lying within a window.

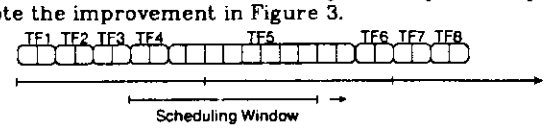
4.2.1. Comments on the Continuous Algorithm

There are several reasons for moving the allocation and scheduling windows in the way described above. In the case of the allocation window, there is no particular advantage in testing a window position if its leftmost slot is occupied: moving the window another slot will eliminate the occupied slot and may add an unoccupied slot at the right end. In addition, there is no point in testing a window position if an unoccupied slot has just been lost off the window's left edge: the best that could have happened is to add another unoccupied slot on the right side, which makes the new window position equivalent to the old position. There is much in common between this method of task force allocation and the "bit-map" method of memory allocation; for example, see [Habermann 76] Chapter 8.

There are similar arguments for moving the scheduling window so that its leftmost slot is always the leftmost slot of a task force. If the leftmost window slot is empty, nothing is lost by moving the scheduling window another slot to the right. If the leftmost slot is occupied but its process isn't the leftmost slot of a task force, then that task force cannot possibly be coscheduled: we might as well advance the window to the next task force. Moving to the leftmost slot of a task force guarantees that every window position coschedules at least one task force and every task force is coscheduled at least once in every sweep.

However, if the scheduling window were simply advanced one task force each time slice, task forces of different size and location would receive unequal treatment. For example, consider the situation of Figure 3 where P is 10 and a task force with 10 processes is surrounded by several task forces with 2 processes. The scheduling window tends to move more slowly across small task forces than across large ones, thereby giving the small ones more coscheduled time slices: in the figure, TF4 receives 4 coscheduled slices to every 1 for TF5. In addition, large task forces cast a "shadow" over the task forces just to their right: although TF4 and TF6 each have two processes, TF4 receives much more coscheduled time than TF6. Finally, task forces at the leftmost end of the process sequence receive harsher treatment than those at the rightmost end (the leftmost end of the sequence casts a shadow equivalent to that of a

task force with P processes). To reduce this discrimination, the scheduling window should be moved each time slice until a) its leftmost process is the leftmost process of a task force, and b) that task force has not yet been coscheduled in the current sweep across process space. Note the improvement in Figure 3.



Task Force	Coscheduled Slices Per Scheduling Sweep: Move Window to Next Task Force	Coscheduled Slices Per Scheduling Sweep: Move Window to Non-coscheduled TF
TF1	1	1
TF2	2	1
TF3	3	1
TF4	4	1
TF5	1	1
TF6	1	1
TF7	2	1
TF8	1	1

Figure 3. An example of unequal treatment of different task forces. The columns indicate how many coscheduled time slices each task force receives each sweep if a) the scheduling window is moved one task force each time slice, and b) the scheduling window is moved each time slice until the leftmost task force is one that hasn't been coscheduled yet this sweep.

The continuous algorithm reduces the problems caused by the rigidity of the row structure in the matrix algorithm. Task forces can be packed more tightly in process space since empty slots that would have been in separate rows under the matrix algorithm may still lie within a single allocation window position. The context swapping load is distributed more evenly by the continuous algorithm than the matrix algorithm because not all processors change scheduling priority each time slice. In spite of its dense packing of task forces the continuous algorithm generally requires less work during allocation than the matrix algorithm; see the simulation results for details.

Unfortunately, the continuous algorithm still suffers from several drawbacks. The most serious problem concerns divisions within task forces. When the process sequence becomes populated with many small "holes" (contiguous empty slots), new task forces are likely to be divided between several holes; the distance in the process sequence between the leftmost and rightmost processes of a new task force, which I call the *width*, may be substantially greater than the size of the task force. A small task force with a large width will have similar scheduling properties to a large task force with the same width, in that there are only a few scheduling window positions for which the task force will be coscheduled. As task forces become split between several holes, even the improved scheduling algorithm discussed above becomes unfair: small contiguous task forces will receive many more coscheduled time slices than those that are large or badly split. This situation is similar to external fragmentation in segmentation systems, where utilization of primary memory degrades substantially if the memory becomes too fragmented. The simulation results verify that the continuous algorithm's behavior deteriorates as process space becomes fragmented.

4.3. The Undivided Algorithm

This algorithm is identical to the continuous algorithm in every respect except that during allocation all of the processes of each new task force are required to be contiguous in the linear process sequence (i.e. the task force may not be divided between two or more holes). The undivided algorithm does not pack process space as efficiently as the continuous algorithm, but reduces fragmentation and thus results in substantially better system behavior under heavy loads (see the simulation results). In fact, this algorithm performed smoothly under a variety of system conditions, and showed the least sensitivity of the three algorithms to factors such as task force size and system load.

5. Analysis of the Algorithms Using Simulation

In order to gain a more quantitative understanding of the algorithms, a simulator has been written. The simulator analyzes the scheduling behavior of a fifty-processor system using each of the above three scheduling algorithms under a variety of synthetic loads.

5.1. The Simulation Model

Because of the scarcity of general-purpose multiprocessors, there exists almost no information on how such systems are likely to be used by concurrent programs. The simulation results are based upon a simple-minded model parameterized in the following way:

Size	The expected task force size (individual task forces are chosen from a pseudo-random exponential size distribution).
Load	The expected ratio of the number of runnable processes to the number of processors (this value is used to compute task force arrivals, where the expected arrival rate is a linear function of load with negative slope).
Lifetime	The expected lifetime of task forces (actual lifetimes are chosen from a pseudo-random exponential distribution).
Idle Fraction	The simulator permits task forces to be in the system without being runnable. The idle fraction is defined as the expected fraction of task forces' lifetime that they are not runnable; task forces are made runnable and not runnable for pseudo-random exponential periods of time based on the idle fraction. Most of the results presented below assume a zero idle fraction; Section 5.4 discusses the effects of idle time.

For lack of a suitable model of interaction between the processes of a task force, the simulator does not allow for a task force to be partially runnable. All the processes of a task force are made runnable or not runnable together. Fortunately, this assumption leads to a pessimistic estimate of coscheduling.[1]

[1] In a real system, if a task force is only partially coscheduled then the executing fragment may well block on communication with the descheduled part of the task force; another task force might become coscheduled by the alternate selection mechanism. In the simulator, the fragment continues to execute even though it is not coscheduled, and hence leads to a lower overall estimate of coscheduling.

Since the purpose of the simulation is to measure how effective the three algorithms are, most of the results are presented in terms of *coscheduling effectiveness*. Coscheduling effectiveness is the mean (across all the time slices of a simulation run) of the ratio of the total number of processors executing coscheduled processes to the total number of processors with runnable processes. A coscheduling effectiveness of 1 is ideal. Coscheduling effectiveness measures the system's ability to coschedule task forces, NOT the response time that will be seen by individual users or the overall performance of the system. In situations where coscheduling is not needed (for example, when there is no interprocess communication), overall system performance may not depend on coscheduling effectiveness.

5.2. Effectiveness as a Function of Load

Figure 4 plots the coscheduling effectiveness of the three coscheduling algorithms as a function of system load, for a mean task force size of 13.5 processes and no idle time. As expected, all of the algorithms performed quite well for underloaded systems and degraded as the system load increased. Above a load of two or three the coscheduling effectiveness leveled off at about two-thirds.

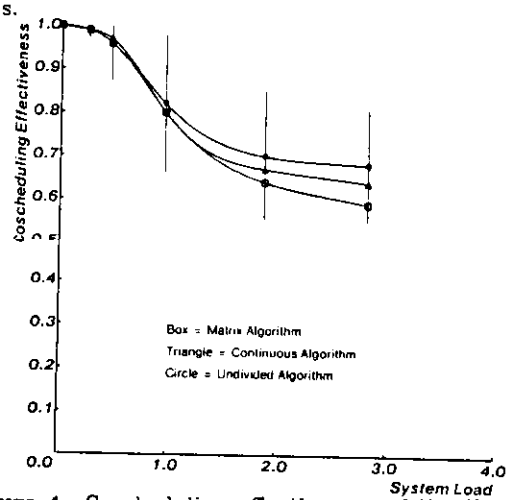


Figure 4. Coscheduling effectiveness of the three algorithms for average task force size 13.5 processes, no idle task forces. The vertical lines are 80% confidence intervals for the undivided curve for individual time slices.

The same overall behavior as in Figure 4 was observed for all of the system conditions simulated here. This can be understood by considering the two factors that result in a coscheduling effectiveness less than 1.0:

Straddling. In the continuous and undivided algorithms, it is possible for a task force to straddle the right end of the scheduling window. If this occurs, then those processes inside the window will execute as a fragment and lower the system's coscheduling effectiveness.

Alternate Selection. In all three algorithms, alternate selection occurs when there are vacant process slots or unrunnable processes in the high-priority portion of process space. Because of its simple-mindedness, alternate selection does not often result in coscheduling. If there are other runnable processes in the processors for which alternate selection occurs, they will probably execute as fragments and degrade the system's coscheduling effectiveness.

For an average system load less than 1.0 almost all processes are allocated in the first row (for the matrix method) or in the first P processes in the sequence (for the other two methods). Thus straddling almost never occurs. When vacant slots exist in the high-priority portion of process space, it is likely that the processors involved contain no processes at all so these holes will not degrade effectiveness. When the average system load becomes greater than 1.0 then both straddling and alternate selection begin to occur and coscheduling effectiveness degrades. Straddling and alternate selection are functions of how task forces are packed into process space; for large loads the packing arrangement becomes independent of load (it depends only on the task force size distribution and the allocation algorithm), so coscheduling effectiveness levels off.

Figure 4 indicates that the algorithms differ in effectiveness by only about 15%. This is not surprising since one of the two contributions to poor coscheduling effectiveness is alternate selection, which is done in the same way by each of the algorithms. The matrix algorithm avoids straddling completely, but incurs slightly more alternate selection as a result. Under heavy loads the undivided algorithm performs between five and ten percent better than the continuous algorithm which in turn performs five to ten percent better than the matrix algorithm.

Figure 4 also plots 80% confidence intervals for the undivided algorithm. The confidence intervals are for individual time slices: in any given time slice one can expect the coscheduling effectiveness to fall within the range of the bars with 80% probability. Note that the short term fluctuations for any single algorithm are larger than the differences between the algorithms. However, in spite of the short-term fluctuations, the average over several time slices converges very quickly. Different runs with different random seeds produced identical average effectiveness values to within several significant digits.

5.3. Effectiveness as a Function of Task Force Size

The differences between the algorithms are most apparent in comparisons of system performance under varying task force sizes. Figure 5 shows how coscheduling effectiveness varies as a function of task force size for an average load of two. For each size, the algorithms showed similar behavior (as a function of load) to that of Figure 4 with variations only in the level at which effectiveness stabilized for high loads.

The data in Figure 5 supports the predictions made earlier. For very small and very large task forces all three algorithms perform quite well: in the limiting cases of mean size 1 or 50 the scheduling effectiveness of each algorithm is 1. The continuous algorithm performs worst when there are many small task forces. Under these conditions the average hole size will be small; task forces are likely to be fragmented between several small holes and hence have widths much larger than their sizes. As the average task force size increases so does the average size of the holes; task force fragmentation occurs less drastically so coscheduling effectiveness improves.

The matrix algorithm is not as prone to fragmentation as the continuous algorithm, so its performance does not suffer as greatly at the hands of small task forces. However, as the average task force size approaches 25 (one half the number of processors) the

matrix algorithm is unable to pack them very densely in the matrix rows. At an average size of 25, process space will only be about 50% packed; since alternate selection does not produce much coscheduling, the coscheduling effectiveness is only about 50%.

The undivided algorithm offers a compromise where large task forces can be packed relatively densely, but small task forces do not cause fragmentation. It shows less sensitivity to task force size than either of the other two algorithms. Since no data is available on what kind of task force size distribution to expect in actual systems the undivided algorithm appears to be the best choice. It seems likely that in multiprocessor systems of the near future there will be more small task forces than large ones (it will be easier to program small amounts of concurrency than large amounts), so the continuous algorithm is especially undesirable. Since the undivided algorithm performed worst with a mean task force size of around 13 processes, that size is used in Figure 4 and in most of the remaining measurements.

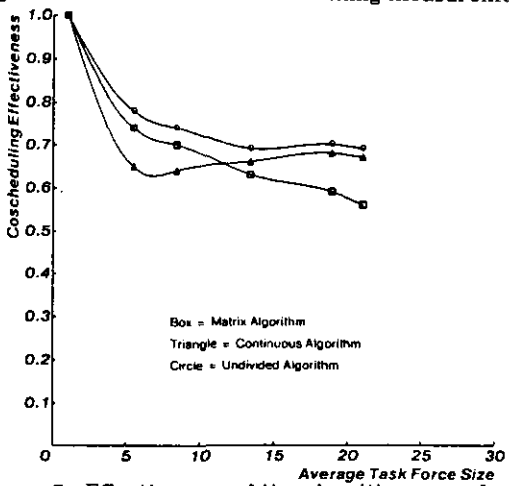


Figure 5. Effectiveness of the algorithms as a function of mean task force size with a system load of 2.0 and no idle task forces.

5.4. Idle Task Forces

In actual systems, all processes cannot be expected to be runnable all of the time. If a task force becomes idle while waiting for some external event such as terminal input, then its processes occupy slots in process space without being runnable. This reduces the degree of coscheduling in the system by causing more alternate selection to occur. Experience with timesharing systems indicates that many programs spend most of their time in an idle state, so the simulator was modified to provide data on the effects of idle task forces.

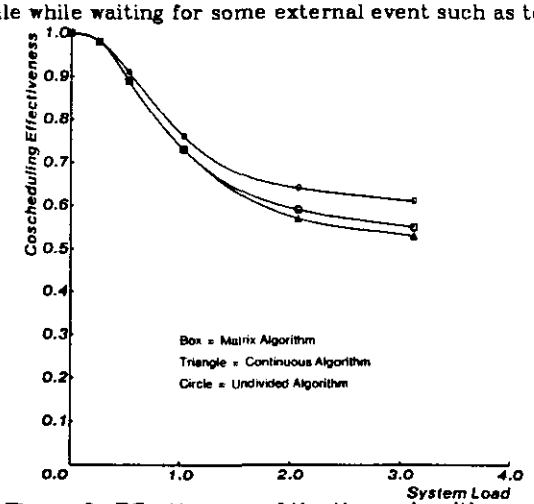


Figure 6. Effectiveness of the three algorithms as a function of system load for an average task force size of 13.5 processes and an idle fraction of .5.

Figure 6 plots coscheduling effectiveness as a function of load for an average task force size of 13.5 and an idle fraction of one half. Although the general behavior of the algorithms is not altered greatly by idle time, a comparison of Figures 4 and 6 shows that the continuous algorithm suffers somewhat more from idle time than the other two algorithms. Figure 7 charts coscheduling effectiveness as a function of idle fraction for an average system load of two. For very large idle fractions the performance degradation is as high as one third. The curves of Figure 7 can be explained in the following way. In the limiting case of very high idle fractions only one task force in the high-priority portion of process space (the current row or scheduling window) will be runnable. If any other task forces are to be coscheduled then that coscheduling must occur as a result of alternate selection. Note also that with an average load of two and an average task force size of 13.5, several task forces are likely to be runnable at any given time. The continuous algorithm tends to fragment task forces such that it is quite likely that the runnable task forces will overlap in their processor usage, hence only one will be coschedulable. Both the undivided algorithm and matrix algorithm tend to allocate task forces in contiguous slots (the undivided algorithm by design, the matrix algorithm by a quirk of its implementation), so there will be fewer overlaps between the runnable task forces. Thus under high idle fractions the matrix and undivided algorithms have somewhat better characteristics than the continuous algorithm.

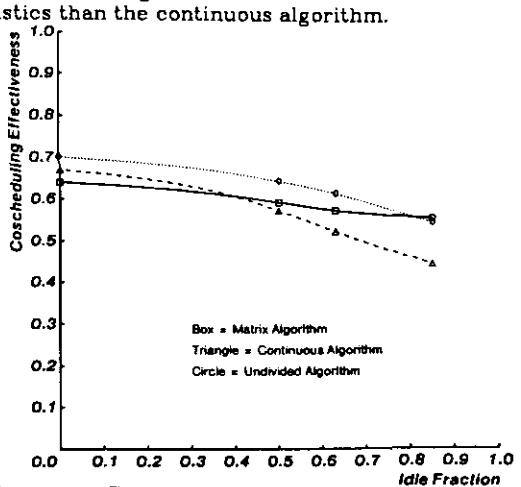


Figure 7. Effectiveness as a function of idle fraction for an average task force size of 13.5 processes and a system load of 2.0.

5.5. Relative Treatment of Different-Size Task Forces

Figure 8 graphs the *coscheduled fraction* as a function of task force size for the undivided algorithm under three different system loads. The coscheduled fraction for a task force is the ratio of the number of coscheduled time slices received by the task force to the total number of time slices when at least one of the task force's processes is executing. Under light loads all task forces are nearly always coscheduled, but under moderate or heavy loads the largest task forces are coscheduled less than one third of the time. Although Figure 8 contains data for just the undivided algorithm, the

corresponding data for the other two algorithms is similar.

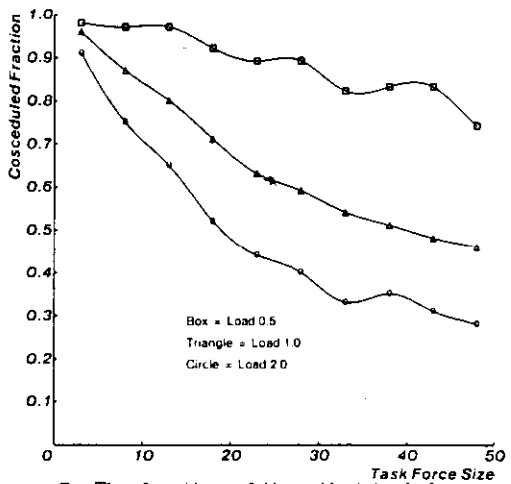


Figure 8. The fraction of time that task forces are coscheduled, as function of task force size. These measurements are for the undivided algorithm with an average task force size of 13.5 processes.

In Figure 9 the fraction of the processes of a task force executing (averaged over the time slices when at least one process of the task force was executing) is plotted as a function of task force size. Once again all task forces receive good, nearly identical, treatment when the system is lightly-loaded and large task forces suffer somewhat more than small task forces as the load increases. However, it is encouraging to note that even under heavy loads the average executing fragment for large task forces contains more than half of the processes of the task force. The vertical bars in Figure 9 are 80% confidence intervals for the behavior of any single task force in the load=1.0 curve.

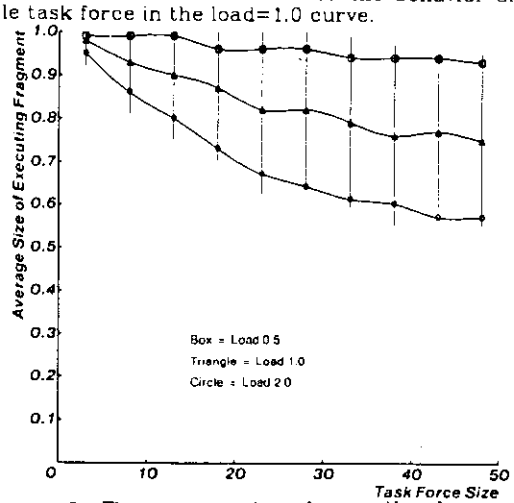


Figure 9. The average size of executing fragments (as a fraction of total task force), measured using the undivided algorithm with an average task force size of 13.5 processes. The vertical lines are 80% confidence intervals for the load=1.0 curve for a single task force of the given size.

Figure 10 shows the relative number of time slices given to each size of executing fragment for large task forces (30-50 processes) using the undivided algorithm under three different system loads. The rightmost point in each curve is the overall coscheduled fraction for task forces with sizes between 30 and 50.

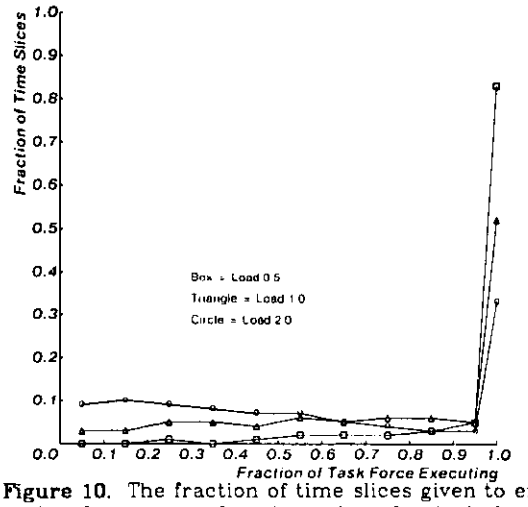


Figure 10. The fraction of time slices given to executing fragments of various sizes for task forces with 30-50 processes, measured using the undivided algorithm with an average task force size of 13.5 and no idle task forces.

5.6. Efficiency of Allocation and Scheduling

The simulator gathered data about the amount of work involved in allocation and scheduling; see Table I. During allocation the undivided algorithm required many more potential task force locations to be checked than either of the other two algorithms. The continuous algorithm consistently required the least number of checks but produced somewhat less coscheduling as a result. The continuous and undivided algorithms provided smoother processor scheduling, with only about half as many context swaps occurring in each time slice compared to the matrix algorithm.

	Matrix Algorithm	Continuous Algorithm	Undivided Algorithm
Average number of positions examined during each allocation	1.6	1.3	8.7
Average fraction of processors context swapping each time slice	.67	.37	.37

Table I. Allocation and scheduling statistics for the three algorithms with a load of 1.9, an average task force size of 13.5, and no idle task forces.

5.7. Summary

The coscheduling of task forces is in many ways very similar to the dynamic allocation of memory. Naive scheduling algorithms can easily lead to thrashing; however, with a little care quite acceptable degrees of coscheduling appear to be obtainable. The algorithms presented here represent three sets of tradeoffs involving the cost of allocation and scheduling, the density of packing (both within a task force and for the system as a whole), and forms of internal and external fragmentation. The continuous algorithm provides fast allocation and scheduling and dense packing of the system as a whole, but is prone to external fragmentation. It consistently performs worse than the other two algorithms. The matrix algorithm provides fast allocation and scheduling, but suffers from internal fragmentation when many large task forces are present. However, its overall performance is not much worse than the undivided algorithm and its implementation is the simplest. The undivided algorithm requires more effort in allocation than either of the other two algorithms, but provides for dense packing both within task forces (they are always allocated contiguously) and for the system as

a whole; this resulted in insensitivity of the algorithm to several conditions that degraded the performance of the other algorithms.

For moderate system loads (around 1.0) a coscheduling effectiveness of between 0.7 and 0.9 can be expected. Application programmers can expect small task forces to be coscheduled almost all the time under almost any conditions; under moderate loads large task forces will be coscheduled about half the time with the average executing fragment containing about four fifths of the processes of the task force.

Some caution must be exercised in interpreting the simulation results, since the algorithms and/or simulation model do not include several factors that could affect performance. As mentioned in Section 5.1, scheduling dependencies between the processes of a task force are not modeled: this simplification make the results presented here pessimistic. Neither the simulator nor the algorithms take into account the possibility that task forces may expand or shrink in size dynamically. The algorithms assume that any process may be assigned to any processor; this may not be a valid assumption in some systems. Finally, the distributions used to model task force size and other parameters could turn out to be inappropriate; to date, no data is available to validate these choices. It is encouraging to note that the simulation results were only mildly sensitive to several factors such as task force size and idle fraction. This suggests that changes in the distributions would have only small effects on the results of the simulations.

6. Conclusions

This paper has shown how scheduling algorithms designed for systems with independent processes break down when they are applied to collections of processes that interact frequently. Naive short-term schedulers result in wasteful context swaps which can be avoided by keeping a process' execution state loaded if it is likely to be waiting for only a short amount of time. Naive long-term schedulers cause process thrashing to occur. This problem is more difficult to avoid; three algorithms for coscheduling were described, and simulation results showed that all three provided substantial degrees of coscheduling, although one algorithm consistently outperformed the others by 10-20%.

Because of its simplicity, the matrix algorithm was implemented in Medusa. Unfortunately, user load on Medusa has been light, so it has not been possible to validate the simulations with the actual system.

7. Acknowledgments Pradeep Sindhu played an important role in the development of the ideas presented here and implemented pauses as part of the Medusa communication mechanisms. Peter Hibbard and Don Scelza participated in early discussions of coscheduling techniques.

8. References

[Denning 70] Denning, P.J. Virtual memory. *Computing Surveys 2*,3 (Sept. 1970), 153-189.

[Habermann 76] Habermann, A.N. *Introduction to Operating System Design*. Science Research Associates, Chicago, IL, 1971.

[Nelson 81] Nelson, B.J. *Remote Procedure Call*. Technical Report, Xerox Palo Research Center, CSL-81-9, May 1981.

[Ousterhout 80] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S. Medusa: an experiment in distributed operating system structure. *Comm. ACM 23*,2 (Feb. 1980), 92-105.

[Ritchie 74] Ritchie, D.M. and Thompson, K. The Unix time-sharing system. *Comm. ACM 17*,7 (July 1974), 365-375.

[Spector 82] Spector, A.Z. Performing Remote Operations Efficiently on a Local Computer Network. *Comm. ACM 25*,4 (April 1982), 246-259.

[Swan 77] Swan, R.J., Fuller, S.H., and Siewiorek, S.P. Cm* -- a modular, multi-microprocessor. *Proc. AFIPS 1977 NCC, 46*, AFIPS Press, Arlington, VA 1977, 645-655.