Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors

Patrick G. Sobalvarro[*]
pgs@lcs.mit.edu

William E. Weihl[†]
weihl@lcs.mit.edu

Abstract

We present demand-based coscheduling, *a new approach to scheduling parallel computations on multiprogrammed multiprocessors. In demand-based coscheduling, rather than making the pessimistic assumption that all the processes constituting a parallel job must be simultaneously scheduled in order to achieve good performance, we use information about which processes are communicating in order to coschedule only these; the result is more opportunities for coscheduling and fewer preemptions than in more traditional coscheduling schemes. We introduce two particular types of demand-based coscheduling. The first is dynamic coscheduling, which was conceived for use on message-passing architectures. We present an analytical model and a simulation of dynamic coscheduling that show that the algorithm can achieve good performance under pessimistic assumptions. The second is predictive coscheduling, for which we present an algorithm that detects communication by using virtual memory system information on a bus-based shared-memory multiprocessor.*

1 Introduction

This paper describes *demand-based coscheduling,* a new approach to scheduling parallel computations on multiprogrammed multiprocessors, which generalizes earlier work presented in [11]. Under demand-based coscheduling, processes are scheduled simultaneously only if they communicate; communication is treated as a demand for synchronization. Processes that do not communicate need not be coscheduled. In particular, demand-based coscheduling is a *dynamic* approach, because it coschedules processes that are currently communicating or have done so recently and thus are expected to do so again in the near future. Because demand-based coscheduling uses more information than does Ousterhout's form of coscheduling [9], it can reduce the difficulty of the scheduling problem and exploit opportunities for coscheduling that traditional coscheduling cannot. Because it does not rely on a particular programming technique, such as task-queue based multithreading, demand-based coscheduling is applicable in domains where process control [13] is not.

Demand-based coscheduling is intended for scheduling mixed loads of parallel and serial jobs, where the parallel jobs may be synchronous message-passing applications, sets of processes communicating using shared memory, or clients and servers communicating through kernel-mediated remote procedure calls. It is also intended for use on a wide variety of platforms that support timesharing: large message-passing multiprocessors, message-passing- and shared-memory-based departmental servers, desktop shared-memory multiprocessors with a low degree of parallelism, and networks of workstations. In the sections ahead, we will describe algorithms for demand-based coscheduling on such platforms.

1.1 Goals

Ousterhout compared parallel scheduling and virtual memory systems in [9]. He suggested that coscheduling is necessary on timeshared multiprocessors running parallel jobs in order to avoid a kind of process thrashing that is analogous to virtual memory thrashing. This kind of process thrashing arises because process can run for only a short period before blocking on an attempt to synchronize with another process that is not currently scheduled; the result is greatly increased numbers of context switches.

We build on this analogy of parallel scheduling to virtual memory, but rather than building a mechanism that resembles swapping, as traditional coscheduling

[*]Support for Patrick Sobalvarro's work was provided in part under a Draper Fellowship from the Charles Stark Draper Laboratory, Inc. This work was supported in part by the Advanced Research Projects Agency under Contract N00014-91-J-1698, by grants from IBM and AT&T, and by an equipment grant from DEC.

[†]Currently on leave at the DEC Systems Research Center, Palo Alto, CA.

does, we seek to produce a mechanism that resembles demand paging. To take the analogy somewhat further, our goals are to produce a scheduler that is *non-intrusive* in the same way that demand paging is non-intrusive: we do not want to impose on the programmer a particular programming model. For example, while demand-based coscheduling could be compatible with a task-queue-based multithreaded approach like process control [13], we do not want to *require* that all parallel applications be coded in a multithreaded fashion in order not to suffer excessive context-switching.

Again as with demand paging, we want an approach that is *flexible*: we wish to free the programmer and the compiler writer from consideration of exactly how many processors are present on the target machine, in the same way that demand paging frees the programmer and the compiler writer from considering exactly how much physical memory is present on the target machine. This is as distinct from traditional coscheduling [9], in which there is no clear means for scheduling jobs with more processes than there are nodes on the multiprocessor.

Finally, we want an approach that is *dynamic*, and can adapt to changing conditions of load and communication between processes. For example, we expect that client/server applications will be particularly important on multiprocessor systems. In such applications, it may not be known ahead of time which client processes will communicate with which servers, but if the rate of communication is sufficiently high, coscheduling will be important. Examples might include SQL front ends communicating with a parallel database engine, or window system clients and servers, or different modules in a microkernel operating system running on a multiprocessor.

Demand-based coscheduling meets these goals: it is non-intrusive, flexible, and dynamic. Like demand paging as compared to swapping, it is a mechanism that improves performance in most cases without requiring extra knowledge from the programmer or the compiler writer.

1.2 Terminology

We use the word *job* to describe a distinct application running on a computer. The application may be a single serial process that does not communicate with any other process but the kernel; or it may be a multithreaded application consisting of separate processes sharing a single address space; it could be a single-process, multiple-data application communicating with message-passing; or it could even be a client/server application consisting of one or more server processes and one or more client processes communicating with each other. The important point is that a job is a logically distinct application consisting of one or more processes that communicate.

We use the word *process* in the traditional way that it is used in the operating systems literature: we mean the state of a serially-executed program with an address space (possibly wholly or partly shared with other processes) and a process control block. A process may be executed on at most one processing node of a multiprocessor at a time.

In order to hide the latency of certain operations, a process may have one or more *threads* of control; these share the address space of the process and its process control block. Depending on the threads implementation, they may be dispatched by the kernel or by a user scheduler, or some combination thereof. Demand-based coscheduling does not require multithreading, but may enhance the performance of multithreaded applications.

2 The Problem of Timesharing Multiprocessors

To date, parallel computers have been used mostly for the solution of scientific and engineering problems, and as testbeds for research in parallel computation.

In these problem domains, the problem of scheduling parallel jobs is simplified. Batch scheduling may be appropriate if the problems are large and the I/O and synchronization blocking rates are low. If the I/O and synchronization blocking rates are low and the multiprocessor has a larger number of nodes than are demanded by the problems, simple space partitioning and a batch queue may be the best choice. Such job-scheduling policies are particularly appropriate for very expensive computers, where economics will dictate careful planning of the job load, and users should be encouraged to perform their debugging off-line, under emulation if possible.

2.1 Problems with batch processing and space partitioning

However, as multiprocessors continue to follow the course set by uniprocessors over the past forty years, they have begun to move out of laboratories and computing centers and into offices. Already multiprocessors with relatively small numbers of nodes (4 – 16) have become popular as departmental servers, and we have begun to see desktop machines with two and four nodes. In business environments, these machines do not typically run explicitly parallel jobs, although we can expect that explicitly parallel compute-intensive jobs will appear once the platforms have become sufficiently popular. Instead, they are purchased because the typical departmental server and many desktop machines run large numbers of processes, and some of the processes are sufficiently compute-intensive that sharing the memory, disk and display resources of the machine is the most economical solution.

In these office environments, with mixed loads of client/server jobs, serial jobs, and (in the future) parallel jobs, the scheduling problem becomes more complex. Batch scheduling is inappropriate, because response times must be low. Simple space-partitioning will not be sufficient in such an environment, because the number of processes will be high compared to the number of processors. Furthermore, it can be difficult to know in advance how many processes a job will require or which processes will communicate with other processes — under a protocol like Microsoft's Object Linking and Embedding (OLE), for example, an editor may communicate with a spreadsheet or a database depending on which document has been loaded.

2.2 Independent timesharing results in poor performance

Crovella *et al.* have presented results in [3] that show that independent timesharing without regard for synchronization produced significantly greater slowdowns than coscheduling, in some cases a factor of two worse in total runtime of applications.[1] Chandra *et al.* have reported similar results in [2]: in some cases independent timesharing is as much as 40% slower than coscheduling. In [5], Feitelson and Rudolph compared the performance of gang scheduling using busy-waiting synchronization to that of independent (uncoordinated) timesharing using blocking synchronization. They found that for applications with fine-grained synchronization, performance could degrade severely under uncoordinated timesharing as compared to gang scheduling. In an example where processes synchronized about every 160μsec on a NUMA multiprocessor with 4-MIPS processing nodes, applications took roughly twice as long to execute under uncoordinated scheduling as they did under gang scheduling.

In general, the results cited above agree with the claims advanced by Ousterhout in [9]: under independent timesharing, multiprogrammed parallel job loads will suffer large numbers of context switches, with attendant overhead due to cache and TLB reloads. The extra context switches result from attempts to synchronize with descheduled processes resulting in blocking. As Gupta *et al.* have shown in [6], the use of non-blocking (spinning) synchronization primitives will result in even worse performance under moderate multiprogrammed loads, because, while the extra context switches are avoided, the spinning time is large.

Although the literature to date has described experiments with relatively small numbers of jobs timesharing a multiprocessor, we may expect (and know, from experience) that departmental servers in practice will be heavily loaded for some portion of their lifetime. The reason is a simple economic one: a system that is not heavily loaded is not fully utilized; an underutilized system is a waste of resources. We may expect that more heavily loaded systems will suffer even higher synchronization blocking rates under independent timesharing, and commensurately higher context switching overhead.

2.3 Traditional coscheduling

We see that on timeshared multiprocessors, some mechanism must be provided to ensure that extra context switch overhead due to synchronization delays is avoided. Ousterhout's solution was *coscheduling,* described in [9]. Under this traditional form of coscheduling, the processes constituting a parallel job are scheduled simultaneously across as many of the nodes of a multiprocessor as they require. Some fragmentation may result from attempts to pack jobs into the schedule; in this case, and also in the case of blocking due to synchronization or I/O, alternate jobs are selected and run.

Relatively good performance has been reported for competent implementations of traditional coscheduling. Gupta *et al.* report in [6] that when coscheduling was used with 25-millisecond timeslices on a simulated system, it achieved 71% utilization, as compared to 74% for batch scheduling (poorer performance is reported with 10-millisecond timeslices). Chandra *et al.* conclude in [2] that coscheduling and process control achieve similar speedups running on the Stanford DASH distributed-shared-memory multiprocessor as compared to independent timesharing.

However, traditional coscheduling suffers from two problems. The first is that, without information about which processes are communicating, it is not clear how to extend any of Ousterhout's three algorithms to work on jobs where the number of processes is larger than the number of processors — the best one might

[1] Crovella *et al.* found that hardware partitions gave the best performance in their experiments, but, as we have discussed above, these are not feasible when one has a large number of jobs to run on a small number of processors.

do would be an oblivious round-robin among the processes during a timeslice in which the job was allocated the entire machine. The second is that the selection of alternate jobs to run, either when the process allotted a node is not runnable or because of fragmentation, is not in any way coordinated under Ousterhout's coscheduling.

We may expect the first problem to become significant as multiprocessors become more prevalent. Manufacturers wishing to provide systems of varying expense and power already vary the number of nodes on the multiprocessors they sell, so that one may buy bus-based symmetric multiprocessors with as few as two or as many as six processors from some manufacturers. The application programmer must then be concerned with somehow keeping the number of processes that constitute a parallel application flexible. This is easy if the application is a multithreaded one using a task queue. But if the application uses a client/server model, or if it consists of independent processes communicating through message-passing or some smaller amount of shared memory, the extra heavyweight context switches required in the case of frequent synchronization will result in considerable overhead.

The second problem is a performance problem. Although the loads examined in the works we have cited have typically been highly parallel ones, many parallel jobs have relatively long sections in which many of the processes are blocked. In these sections alternate processes must be selected to run on the nodes where the blocked processes reside. Additionally, the internal fragmentation in Ousterhout's most popular algorithm (the matrix algorithm) results in some nodes not having processes assigned to them by the algorithm during some timeslices; these nodes will also need to perform this "alternate selection." Unfortunately, traditional coscheduling presents no means of coscheduling these alternates. The result is that even in the two-job case examined by Crovella *et al.* in [3], when approximately 25% of the cycles in the multiprocessor were devoted to running alternates, their use decreased the runtime of the application to which they were devoted only about 1%.

2.4 Distributed hierarchical control

Distributed hierarchical control was presented by Feitelson and Rudolph in [4]. The algorithm logically structures the multiprocessor as a binary tree in which the processing nodes are at the leaves and all the children of a tree node are considered a partition. Jobs are handled by a controller at the level of the smallest partition larger than the number of processes required by the job. The placement algorithm strives to balance loads and keep fragmentation low.

Unlike Ousterhout's coscheduling, distributed hierarchical control has a mechanism for the coordinated scheduling of alternates. Suppose K of the nodes allocated to a job cannot run the job's processes, because these processes are blocked. Then the placement algorithm will attempt to find a job with K or fewer processes to run on these K nodes.

If a partition holds processes belonging to different parallel jobs, then the parallel jobs are gang-scheduled within the partition. Distributed hierarchical control thus strikes a middle ground between space-partitioning and coscheduling. It is particularly attractive for larger multiprocessors, where it removes the bottleneck inherent in the centrally-controlled traditional coscheduling of Ousterhout. However, distributed hierarchical control was not designed for smaller machines, such as the desktop machines and departmental servers we have described, on which we expect that it would suffer from the same problems as traditional coscheduling.

2.5 Process control

Tucker and Gupta suggested in [13] a strategy called *process control*, which has some of the characteristics of space partitioning and some of the characteristics of timesharing. Under process control, parallel jobs must be written as multithreaded applications keeping their threads in a task queue. The scheduler divides the number of processors on the system by the number of parallel jobs to calculate the "number of available processors." The system dynamically makes known to each parallel application the number of available processors, and the application maintains as many processes as there are available processors. The processes simply dequeue threads from the application's task queue and run them until they block, at which point they take another thread. If more parallel jobs exist than there are processors, the scheduler timeshares processor sets among the parallel jobs.

One advantage of this approach is that when the processes of a parallel job switch among threads, the switch performed is a low-overhead one that does not cross address-space boundaries, because the multiple threads of an application share an address space. Thus fewer heavyweight context switches need be performed. Tucker and Gupta also cite as an advantage what they call the *operating point effect* — the fact that many parallel jobs will run more efficiently on a smaller number of nodes than on a larger number of nodes, due to the overhead of communication among larger numbers of processes.

Several published works [2, 6, 12] cite good performance for process control, but these works also find that coscheduling can be modified to have equivalently good performance.

It will be clear in what follows that demand-based coscheduling is not at all incompatible with a multithreaded approach; it might even be made to work with process control. But we find process control *alone* to be insufficient for the office environment we have described for two reasons: the requirement that applications be programmed in a particular way, and the high variability of runtimes of memory-intensive applications.

We have already discussed the first problem, that of intrusiveness, to some extent above. For many parallel applications, especially data-parallel applications, a multithreaded approach is entirely appropriate. But for others, applications composed of subtasks that perform distinct and logically autonomous functions, the multithreaded approach may be inappropriate or even impracticable. Examples might include clients and servers that require high rates of communication, but where for security reasons the client is not allowed access to all of the server's data.

Thus process control alone is insufficient as a scheduling approach in the environment we have described, because in requiring that all parallel applications be coded in a task-queue multithreaded fashion, it would require that an important abstraction be given up by the programmer in order to achieve good performance: the abstraction of a process with its own address space. But processes offer modularity and security, and application writers will be loath to give up these qualities in applications where the process abstraction is the natural one.

The second problem, that of high variability of runtimes for some sorts of processes under process control, results from certain parallel jobs requiring more resources than are available on a single node in order to execute efficiently. Under process control, the arrival of new jobs into the system can cause the "number of available processors" to fall below a critical level at which the performance on some jobs will begin to deteriorate worse than linearly.

This implies that in fact the jobs in question show superlinear speedup. In fact this is true in two examples in published works on process control. In [6], the LU application is found to perform very poorly under process control when run on three processors, and the authors point out that a drastically increased cache miss rate is to blame. Similarly, in [2], the Ocean application suffers a twofold decrease in efficiency when run on eight processors as compared to when it is run on sixteen processors. Some of this decrease in efficiency is attributed by the authors to data distribution optimizations being performed in the sixteen-processor case, but not in the eight-processor case. The implication is that, if the data distribution optimizations had not been performed in the sixteen-processor case, the Ocean application would have performed nearly as inefficiently in the sixteen-processor case as in the eight-processor case. So far as one can tell from the published work alone, this attribution of cause may be mistaken, because the same work shows a coscheduling experiment in which data distribution optimizations were not performed. In this experiment, coscheduling among two jobs suffered only a five-percent decrease in efficiency compared to the standalone sixteen-processor case with data distribution optimizations — thus it seems that we can bound above the effect of data distribution optimizations by five percent. Because the authors state that Ocean has a larger working set than the other applications tested, we suspect that the actual cause of the inefficiency here may be the larger number of cache misses that result from the application being executed on a collective cache of half the size as in the sixteen-processor case.

Helmbold and McDowell have documented this sort of "superunitary speedup due to increasing cache size" in [7]. Because of this property of certain parallel applications, their ideal "operating point" is larger than one — possibly considerably larger than one. Thus forcing them to run on fewer processors will be very inefficient. This is not a problem under coscheduling, because under coscheduling the arrival of new jobs does not cause fewer processors to be devoted to the execution of a parallel job.

We believe that the phenomenon of increasing inefficiency with higher loads under process control may be an important problem in practice. This is because software tends to perform near the memory boundaries available on most users' processors. The reason for this pressure is simply economic: purchasers of computer hardware will tend to buy as little memory as possible while still maintaining satisfactory performance on applications; to purchase more would be wasteful. Purveyors of software tend to use more memory to add new features to their applications in order to gain competitive advantage. Programming so as to conserve memory requires more effort and thus costs more, and will be done only insofar as is necessary to keep customers happy.

This pushing at the boundaries of available memory will probably mean that many commercial applications will show superlinear speedup. If process control as it is described in [13] were used as the only means of timesharing a multiprocessor, we would expect that such applications would show poor performance when the job load was high.

3 Demand-based Coscheduling

Demand-based coscheduling is what we call our new approach to scheduling mixed workloads on multiprogrammed multiprocessors. The approach dictates only that processes that are communicating be coscheduled. In particular, demand-based coscheduling does not require a particular method of process placement; processes may be placed or migrated in whatever fashion seems appropriate for load-balancing or data-distribution reasons.

We present two methods for doing demand-based coscheduling, although there might of course be many more. These two methods are *dynamic coscheduling* and *predictive coscheduling*.

Dynamic coscheduling was called *adaptive gang scheduling* in an earlier work [11], and is an approach suited for use on message-passing multiprocessors or on distributed shared-memory multiprocessors in which cache-line-invalidation events can interrupt the processor. Under dynamic coscheduling, messages arriving at a node, if addressed to a process other than the one currently running, sometimes cause preemption of the running process in favor of the process to which the message is addressed. Thus, processes on different processors that communicate frequently will tend to be coscheduled, reducing the amount of context switching and the amount of blocking due to synchronization.

Predictive coscheduling can be used on message-passing or shared-memory multiprocessors, but could also be used on bus-based shared-memory multiprocessors. Under predictive coscheduling, the recent history of communication between processes is used to identify a set of *correspondents* for each process. When a process is scheduled on one node, an attempt is made to schedule its correspondents on other nodes for simultaneous execution.

We can see right away that demand-based coscheduling will be able to perform coscheduling in some cases where traditional coscheduling cannot. For example, consider a case in which two parallel jobs, A and B, are run on an eight-node bus-based shared-memory multiprocessor, as shown in Figure 1. Suppose that the processes labeled '$A1$' are intercommunicating in the current phase of the computation, and the processes labeled '$A2$' are also intercommunicating, but that these two groups do not currently communicate with each other, despite being part of the same application. Suppose also that the processes of job B intercommunicate.

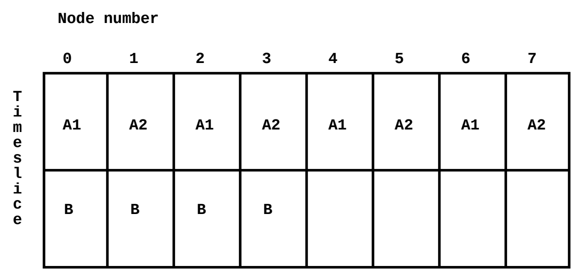

Figure 1: A scheduling scenario in which demand-based coscheduling can achieve better results than traditional coscheduling.

In this case, under both traditional coscheduling and demand-based coscheduling, in the first timeslice, the processes of job A can be run. In the second timeslice, when job B runs, under traditional coscheduling, alternates must be selected to run on the nodes that B's processes do not occupy. Because there is no means of coscheduling alternates in traditional coscheduling, it is highly probable that some mixture of the processes labeled '$A1$' and those labeled '$A2$' will be scheduled, whereas under demand-based coscheduling on a bus-based shared-memory multiprocessor, one of the intercommunicating sets may be picked and run.

4 Dynamic Coscheduling

Dynamic coscheduling identifies communicating processes by examining messages being received in the normal course of the computation. Under dynamic coscheduling, sometimes a message arriving at a node will cause it to start running a process belonging to the application that was running on the processor where the message originated.

On a message-passing multiprocessor, this is straightforward to implement; an arriving message not addressed to the currently running process can trigger an exception (as might be necessary in any case to enforce protection). Alternatively, if protection is not an important issue and the network interface is manipulated directly in user mode, the detection of an arriving message not addressed to the currently running process can be performed by a library routine which can execute a system call in the case when a scheduling decision must be made.

Demand-based coscheduling should also work on many distributed-shared-memory multiprocessors. In a cache-coherence scheme such as the software schemes presented by Chaiken *et al.* in [1], cache line invalidations can be treated in the same fashion as arriving messages. We can do even better on systems with network interface processors, such as FLASH [8] or Typhoon [10]. In these systems, some of the scheduler state can be cached in the interface processor, so that the scheduling decision can be made without consulting the computation processor. The computation processor could be interrupted only when a preemption was needed. In this case the number of exceptions could be kept to the minimum necessary.

It is more difficult to envision applying this scheme to a shared-memory multiprocessor with hardware-only cache-coherence protocols; for such processors predictive coscheduling will be more appropriate.

We now develop a dynamic coscheduling algorithm by taking the simplest possible implementation of this idea and successively modifying it to achieve fair scheduling while maintaining good coscheduling.

4.1 The “always-schedule” dynamic coscheduling algorithm

The first version of the dynamic coscheduling algorithm is the simplest possible one, in which the job for which the arriving message was destined is always immediately scheduled. We have modeled this case analytically with a Markov process for two symmetric jobs of N processes running on N nodes, using the weak assumptions that messages are uniformly addressed, that the processes generating them are memoryless, and that the run-time of processes before they block spontaneously is exponentially distributed. We call the assumptions “weak” because we expect that real processes exhibit greater regularity that would in fact improve the performance of such a scheduler.

The two-job Markov process is a skip-free birth-death process, and a closed-form solution for the steady-state probabilities is possible. The multiprocessor has N nodes. The states of the process are defined as follows: in state i, $N-i$ nodes are running the first job and i are running the second job. If we call the jobs job A and job B, in our model we make use of the quantities q_{SA} and q_{SB}, the rates of spontaneous context switching of processes for jobs A and B. The spontaneous switching rate is intended to capture at once the notion of timeslice expiration and blocking due to I/O or synchronization requirements. A node running a process will switch from running it to the next resident process at this rate. We also use the quantities q_{MA} and q_{MB}, the rates of message-sending for processes of jobs A and B — these are the rates at which the running processes generate uniformly-addressed messages to other processes that make up their jobs.

In summary, then, state 0 is the state in which all the nodes are running job A and no nodes are running job B. In state 64, all the nodes are running job B and no nodes are running job A. In state 32, half of the nodes are running each job.

The steady-state probabilities are then given by

$$p_k = p_0 \prod_{i=0}^{k-1} \frac{(N-i)q_{SB} + i\frac{N-i}{N}q_{MB}}{(i+1)q_{SA} + (N-i-1)\frac{(i+1)}{N}q_{MA}} \qquad (1)$$

where

$$p_0 = \frac{1}{1 + \sum_{k=1}^{N} \prod_{i=0}^{k-1} \frac{(N-i)q_{SB} + i\frac{N-i}{N}q_{MB}}{(i+1)q_{SA} + (N-i-1)\frac{(i+1)}{N}q_{MA}}} \qquad (2)$$

Results for this case are shown in Figure 2. Here we have taken $q_{SA} = q_{SB} = Q_S$ and $q_{MA} = q_{MB} = Q_M$. The vertical axis is steady-state probability. The deep axis is $log_{10}(Q_s/Q_m)$. The horizontal axis along the front gives state number.

Towards the front of the graph, we see that the probabilities of being in the states where all the nodes are running one job or the other are high, and the probabilities of being in states where some nodes are running one job and some running the other are low. We see then that the ratio of the rate of sending messages to the rate of spontaneous switching of processes determines the steady-state probability that all processors in the modeled system are running a single job. We found that if several hundred or more messages are sent on average between the spontaneous context switches, then the steady-state probability that all processors are either running one job or all processors are running the other job is about one-half. If fewer messages are sent between spontaneous context switches, then a binomial behavior begins to emerge, so that when only one message is being sent on average between spontaneous context switches, about half of the processors are running one job, and half running another. It is to be noted, though, that when very few messages are being sent, coscheduling is unlikely to be important.

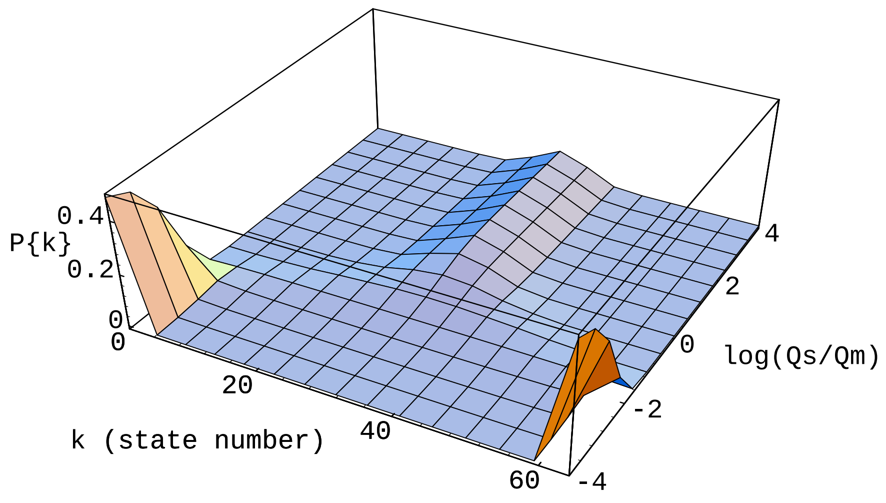

Figure 2: Steady-state probabilities found using a Markov model to calculate dynamic coscheduling performance on a 64-processor system running two jobs. See the text for further details.

It is encouraging that with such a simple rule, we find strong coscheduling behavior under such weak assumptions. Unfortunately, this coscheduling algorithm has a fatal flaw. The flaw is that it is completely unfair, tending to very strongly favor jobs that send a lot of messages. Also, even if message-sending rates are equal, this algorithm may take a very long time to switch out of a state in which most processors are running one job, although this dynamic behavior of the algorithm cannot be seen from the steady-state probabilities alone.

Figure 3 illustrates the unfairness of this algorithm, showing the steady-state probabilities for the case where $q_{SA} = q_{SB} = 0.005$ but $q_{MA} = 0.49$ and $q_{MB} = 0.5$. It can be seen that, despite the fact that the message-sending rates are very close, job A achieves full coscheduling only about 2% of the time whereas job B achieves full coscheduling about three times as often.

4.2 The "equalizing" dynamic coscheduling algorithm

We modified the "always-schedule" dynamic coscheduling algorithm to require that runnable processes receive equal shares of the CPU, within some constant difference. We called this policy "run-time equalization." Because it was more difficult to analytically model the new algorithm, we wrote a discrete event simulator for it, and ran experiments in which we modeled a 64-node multiprocessor running for 100,000 scheduler cycles.

We maintain for each process i a quantity r_i, the number of scheduler cycles for which it has run since the process that most recently joined the scheduler run queue started running. We define a global quantity h, which can be modified to affect the "volatility" of scheduling: a larger value of h causes the scheduler to take longer to switch due to arriving messages.

Run-time equalization works as follows: when a message destined for process j arrives at its node, which is running process i, $i \neq j$, we switch to process j if and only if $r_j + h < r_i$, that is, if and only if process j *lags* process i by more than h scheduler cycles. This definition of h means that if the system is run for no more than H scheduler cycles, and $h = -H$, the "equalizing" algorithm will always behave the the same as the "always-switch" algorithm. This is because r_j cannot be greater than H if the system is run for no more than H cycles, and so necessarily $r_j + h \leq 0$, and in this scenario process i has run for at least 1 scheduling cycle. With this very negative value of h, then, the scheduler will always context-switch due to arriving messages.

On the other hand, if $h = H$ and the system is run for no more than H cycles, a process i will never accumulate more than H scheduling cycles, and it will always be the case that $r_j + h \geq r_i$ (until possibly the H^{th} cycle, when the experiment ends). Thus with this large positive value of h, the scheduler will never context-switch due to arriving messages.

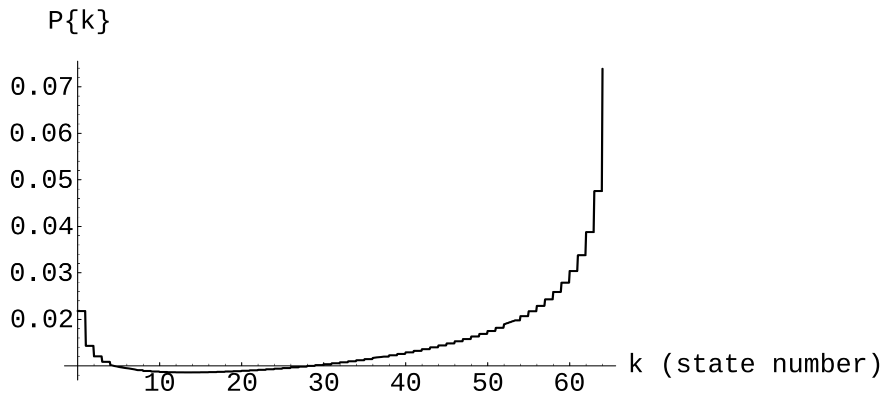

Figure 3: Steady-state probabilities in the case where message-sending rates differ very slightly – see the text for further details.

We found that, for values of h near $-1,000$, reasonably fair performance was attained over the running of an experiment; however, little coscheduling was achieved. The results for the more radical case of message-sending rates of .25 and .5 may be seen in Figure 4.

Our intuition about the failure to coschedule under simple equalization is that, by disregarding more opportunities to coschedule processes, we caused more thrashing. In general, the higher the value of h, the less coscheduling was achieved. One possible solution was to further reduce h, but in fact, we already had a mechanism that proved to work better in practice at recovering strong coscheduling behavior, by ensuring that the scheduler makes progress from job to job.

4.3 The "epochs and equalization" dynamic coscheduling algorithm

Consider a scenario in which about half of the nodes on a multiprocessor are running one parallel job, and half the other. In our simulation, when a node running parallel job A spontaneously switches to parallel job B, there is a probability of close to $1/2$ that the next message it receives will be destined for a process belonging to job A, provided that message-sending rates for the two jobs are equal. Thus there is a substantial probability that the node will switch quickly back to job B without job A ever having achieved full coscheduling. This probability is greater if switching is mostly spontaneous.

Epoch values are used to reduce this sort of thrashing. The epoch value is maintained in a counter at each node. The counter is incremented at each spontaneous context switch. When a node sends a message, the epoch value is included in the message; when receiving a message, the node considers switching only if the equalization criteria are met and the epoch number is higher than its own. If the node does switch processes, it adopts the higher epoch number as its own.

The result is that nodes "defecting" from a parallel job will not return to the job due to messages being sent by nodes remaining with the job. Progress must be made to the new job before the node will consider switching back. The results for this strategy can be seen in Figure 5, and are quite encouraging — coscheduling behavior is achieved for more than about 300 messages per timeslice, even given our pessimistic assumptions. As in Figure 4, message-sending rates of .25 and .5 are used.

5 Predictive Coscheduling

If we wish to implement coscheduling on a bus-based shared memory multiprocessor, with hardware-only cache-coherence protocols, the detection of communication becomes more complicated than on message-passing architectures. If the program uses library routines for heavyweight remote procedure calls or for semaphores, the invocation of the kernel to deliver messages, perform blocking tests of semaphores, or set semaphores will allow the scheduler to be aware of communication between processes, and dynamic coscheduling can be used.

But if instead processes communicate only through shared memory pages in user mode, the kernel is not invoked, and cannot detect communication when it happens. We might consider using memory protec-

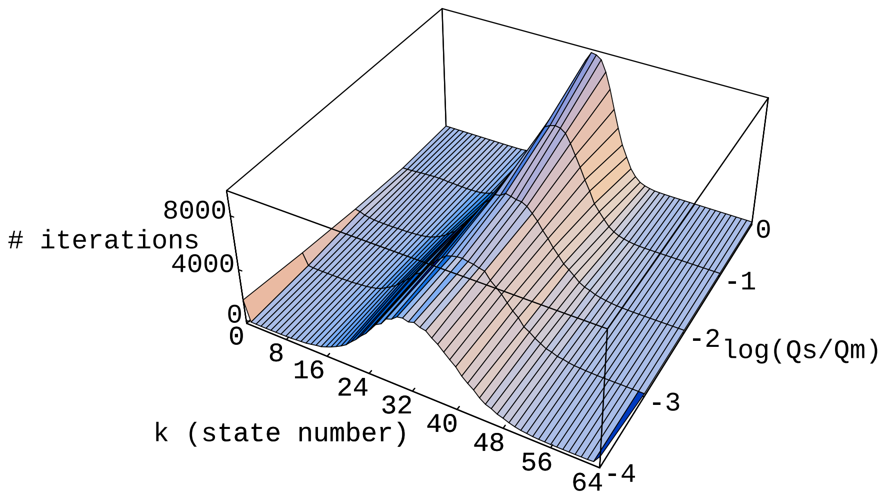

Figure 4: Degree of coscheduling in the case where message-sending rates differ by a factor of two, but equalization is used. The vertical axis approximates steady-state probability – the scale is the number of iterations out of 100,000 in which the process was found in the indicated state.

tion on shared memory pages to signal the kernel the first time during a process's lifetime that it requests access to a shared memory page, but this could be quite expensive if a large number of shared memory pages are touched and memory protection traps are slow.

Another possibility is to recognize that coscheduling is a performance optimization, and is not required for correctness, so that it is feasible to use a mechanism that simply provides hints as to which processes are likely to communicate with each other. If such a mechanism is correct with high probability, it will be sufficient to allow good performance.

We proceed by describing predictive coscheduling in the next section, and then proceed to describe an inexpensive mechanism for detecting communication using virtual memory hardware.

5.1 Correspondents

Under *predictive coscheduling*, processes that have recently communicated with each other are called *correspondents*. As in LRU demand paging, past behavior is treated as a predictor of future behavior, and so predictive coscheduling works by coscheduling runnable correspondents. In particular, when a process is scheduled on a node, an attempt is made to simultaneously schedule on other nodes its runnable correspondents. On a message-passing multiprocessor, this could be done by sending messages to the nodes on which the correspondents resided. On a bus-based shared-memory processor, other processes would be selected for preemption, interrupts would be signalled on their nodes, and the correspondents would be scheduled.

We have not yet tested this strategy, although it appears promising. Clearly a runtime equalization mechanism would be necessary to ensure fairness; possibly a mechanism like epochs would be desirable to reduce thrashing. The selection of processes for preemption is another open question. It might be desirable to select for preemption the processes with the fewest correspondents, because such processes will be runnable in the future under a wider variety of circumstances.

It is also worth noting that if communication between processes is entirely memoryless — so that one pair of processes that have recently communicated is no more likely to communicate in the future than is any other pair of processes — predictive coscheduling will not perform well. This is because predictive coscheduling attempts to predict future behavior on the basis of past behavior, a strategy that will work no better than random selection with uniform probability for memoryless processes. Of course, this scenario is unlikely to arise in most parallel jobs, where the constituent processes will communicate with each other repeatedly.

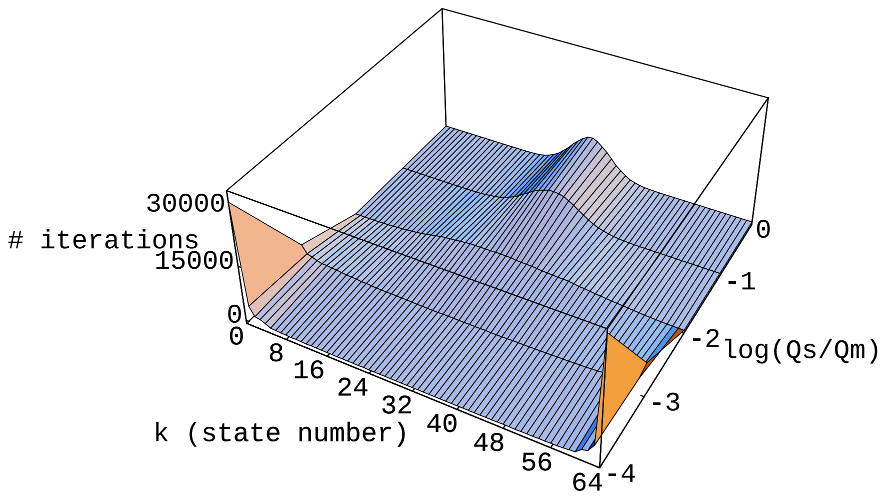

Figure 5: Degree of coscheduling achieved in the case where message-sending rates differ by a factor of two, and both equalization and epochs are used – the vertical axis approximates steady-state probability, being the number of iterations out of 100,000 in which the simulator was in the indicated state.

5.2 Detecting communication through shared memory on bus-based shared-memory multiprocessors

It remains to describe a means of identifying correspondents on bus-based shared-memory multiprocessors — that is, detecting communication through shared memory. Because, as we noted above, we do not think that this information need be perfect, we propose using the information in translation lookaside buffers (TLBs) on processors where these are readable. Processor-readable TLBs are becoming more common, because of the attractiveness of handling TLB misses in software on RISC processors. Among the processors that have readable TLBs are MIPS processors, DEC Alpha processors, and HP PA-RISC processors. Intel 486-series processors also provide a means of reading the TLB through the test instruction interface; reading the TLB does not require disabling virtual memory.

The algorithm for finding correspondents is simple. In the process control block (PCB) for each user process that shares memory, we maintain a field that contains a list of correspondents. For each user process we maintain in memory a data structure, keyed by virtual page address, that contains an entry for each of the shared memory pages mapped in the process. The structure is called the Shared Page Recent Accessors Table (SPRAT). Each entry in the SPRAT contains a list of processes that have recently accessed the page. At certain points in the execution of a process, we iterate over the TLB, finding entries for shared data pages in this process's address space. For each such entry, we find the corresponding page entry in the SPRAT and add the current process to the list there. Then we add the processes in the SPRAT entry to the correspondents list in this process's PCB.

Because the TLB is typically small (256 or fewer entries), it can be searched quickly. Because the replacement policy is typically approximate LRU within a set (or simply LRU on fully-associative TLBs), the TLB contains information about which pages have been read or written recently. We may choose to search it just before descheduling a process, or perhaps also at other convenient times, such as system calls and exceptions.

Entries will also need to be cleared from the SPRAT and the correspondents list in the PCB. One inexpensive possibility is to maintain the lists as FIFOs of fixed and limited size — for example, the number of nodes on the machine would be a good limit. Another possibility is simply to clear the information at pseudorandom intervals — this can also be implemented inexpensively.

There are likely to be other means of detecting communication through shared memory on bus-based shared-memory multiprocessors using virtual memory information — as has been found in the field of lifetime-based garbage collection, the information maintained by a virtual memory system is very rich.

6 Conclusions

We have presented demand-based coscheduling, a new approach to scheduling parallel computations on multiprogrammed multiprocessors that promises better performance by coscheduling only those processes that communicate with each other. Demand-based coscheduling is:

- Non-intrusive — the programmer is not required to write parallel programs in a particular style. *E.g.*, multithreading is not required; if full-fledged processes are a better abstraction, they can be used instead. Process placement or migration algorithms are not imposed by demand-based coscheduling.

- Flexible — If a job composed of a large number of processes is run on a multiprocessor with a small number of nodes, demand-based scheduling can take advantage of local communication patterns that may provide better performance.

- Dynamic — Newly-initiated communication between processes is detected automatically as a demand for synchronization by demand-based coscheduling. Thus it is well-suited to newer programming paradigms (*e.g.*, OLE) that may result in fine-grained communication between processes the programmer could not have anticipated would communicate.

- Decentralized — scheduling decisions are made locally in demand-based coscheduling. Unlike in traditional coscheduling, there is no "alternate coscheduling problem," because there is no centrally-imposed notion of a single currently-scheduled job.

Additionally, because demand-based coscheduling does not limit the number of processes making up a parallel job, conditions of high load will not cause jobs to suffer from the anomalous behavior on memory-intensive programs reported in work on process control.

We presented two forms of demand-based coscheduling: dynamic coscheduling, which is suited for use on message-passing processors and distributed-shared-memory processors with software cache-coherence protocols; and predictive coscheduling, which we expect to work well on shared-memory processors with hardware-only cache-coherence protocols.

We presented analytical and simulation results that show that the number of messages sent per timeslice is a key factor in achieving good coscheduling behavior under dynamic coscheduling, and that with a mean communication rate of more than ~ 300 messages per timeslice in our simulations, strong coscheduling behavior was achieved. We also showed that even under very pessimistic assumptions, dynamic coscheduling can achieve strong coscheduling behavior while maintaining fairness in scheduling. We expect that real applications will achieve even stronger coscheduling behavior.

We discussed a means of using virtual memory hardware to identify correspondents — communicating processes — on shared-memory platforms with hardware-only cache-coherence protocols.

We are currently completing a more faithful simulator that will allow us to evaluate more demand-based coscheduling strategies under a variety of synthetic loads. Following this, we plan to implement demand-based coscheduling on an actual multiprocessor so that it may be evaluated under actual application loads.

Acknowledgements

The authors wish to thank Bert Halstead, Butler Lampson, and Larry Rudolph for valuable discussions of the ideas presented in this paper. Paul Barton-Davis, Allen Downey, Shel Kaphan, and Volker Strumpen were kind enough to read drafts of the paper and give us their detailed comments. Finally, the suggestions of the anonymous referees were helpful and very much appreciated.

References

[1] Chaiken, D., Kubiatowicz, J., and Agarwal, A. "LimitLESS Directories: A Scalable Coherence Scheme," in *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991, pp. 224–234.

[2] Chandra, R., *et al.* "Scheduling and Page Migration for Multiprocessor Compute Servers," in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, San Jose, California, October, 1994, pp. 12–24.

[3] Crovella, M., *et al.* "Multiprogramming on Multiprocessors," in *Third IEEE Symposium on Parallel and Distributed Processing,* 1991, pp. 590–597.

[4] Feitelson, D. G., and Rudolph, L. "Distributed Hierarchical Control for Parallel Processing," in

IEEE Computer, Vol. 25, No. 3, pp. 65–77, May, 1990.

[5] Feitelson, D. G., and Rudolph, L. "Gang Scheduling Performance Benefits for Fine-Grain Synchronization," in *Journal of Parallel and Distributed Computing,* Vol. 16, No. 4, pp. 306–318, December, 1992.

[6] Gupta, A., Tucker, A., and Urushibara, S. "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," in *Proceedings of SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* May, 1991, pp. 120–132.

[7] Helmbold, D. P., and McDowell, C. E. "Modeling Speedup(n) Greater than n," in *IEEE Transactions on Parallel and Distributed Systems,* Vol. 1, No. 2, April 1990, pp. 250–256.

[8] Kuskin, J. *et al.* "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual Symposium on Computer Architecture*, Chicago, Illinois, April, 1994.

[9] Ousterhout, John K. "Scheduling Techniques for Concurrent Systems," in *Third International Conference on Distributed Computing Systems,* October, 1982, pp. 22–30.

[10] Reinhardt, S. K., Larus, J. R., and Wood, D. A. "Tempest and Typhoon: User-level Shared Memory," in *Proceedings of the 21st Annual Symposium on Computer Architecture*, Chicago, Illinois, April, 1994.

[11] Sobalvarro, P. G. "Adaptive Gang-Scheduling for Distributed-Memory Multiprocessors," in *Proceedings of the 1994 MIT Student Workshop on Scalable Computing,* MIT Laboratory for Computer Science Technical Report No. 622, July, 1994.

[12] Tucker, A. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors.* Stanford University Department of Computer Science Technical Report CSL-TR-94-601, November, 1993.

[13] Tucker, A. and Gupta, A. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 159–186.