Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors

Andrew Tucker and **Anoop Gupta**
Department of Computer Science
Stanford University, Stanford, CA 94305

Abstract

Shared-memory multiprocessors are frequently used in a time-sharing style with multiple parallel applications executing at the same time. In such an environment, where the machine load is continuously varying, the question arises of how an application should maximize its performance while being fair to other users of the system. In this paper, we address this issue. We first show that if the number of runnable processes belonging to a parallel application significantly exceeds the effective number of physical processors executing it, its performance can be significantly degraded. We then propose a way of controlling the number of runnable processes associated with an application dynamically, to ensure good performance. The optimal number of runnable processes for each application is determined by a centralized server, and applications dynamically suspend or resume processes in order to match that number. A preliminary implementation of the proposed scheme is now running on the Encore Multimax and we show how it helps improve the performance of several applications. In some cases the improvement is more than a factor of two. We also discuss implications of the proposed scheme for multiprocessor schedulers, and how the scheme should interface with parallel programming languages.

1 Introduction

The computing environment we consider in this paper is that of a multiprogrammed shared-memory multiprocessor, with multiple simultaneously running parallel applications. There are several commercially available parallel systems that support such an environment, including machines from Alliant, Encore, and Sequent. In these environments, where the number of running applications is continuously changing, the issue arises of how an application should maximize its performance. We have observed that the performance of parallel applications degrades when the number of processes associated with all applications exceeds the total number of available processors. One possible reason for the degradation in performance is the preemption of processes while they are executing within a spinlock-controlled critical section. In such a situation, other processes may be wasting processor resources just spinning, waiting to enter that critical section. Other possible reasons for degradation include the overhead of unnecessary context switching, and the problem of processor cache corruption when several processes are being multiplexed on a given processor. Consequently, we would like the number of runnable processes in all applications not to exceed the number of processors in the machine.

Ensuring that the number of processes does not exceed the number of processors is trivial when there is only a single user, as he can make sure that the application spawns the right number of processes. A variation that is allowed by some machines and operating systems is that the user can ask the OS for some number of processors. If that many are available, the OS grants them to him, giving him the appearance of a personal machine with that many processors. For example, a BBN Butterfly can be divided into multiple independent clusters [2]. Many of the message-passing hypercube computers also work this way [1]. However, such a scheme is potentially wasteful of resources, since applications will not be able to utilize processing resources that may become available if the system load decreases at a later time. Ideally, we would like an application to be able to use all processors in the system if it is the only application running. As more applications start running, the number of processors available for this application decreases. Correspondingly, the application should reduce its number of runnable processes. Similarly, when other applications finish and more processors become free, the application should be capable of increasing its number of runnable processes to use the extra processors.

In this paper we present preliminary results from a system that permits control of the number of runnable processes as described above. The system runs on an Encore Multimax under the UMAX operating system. The parallel applications are written using the Brown University Threads package [6]. We have modified the threads package so that it contains the code for controlling the number of runnable processes. This has been done in a way such that the process control is totally transparent to the applications programmer. In addition, we have implemented a server that communicates with the OS kernel to determine how many processes each application should have. The applications periodically communicate with the server to check how many processes they should have runnable, and at suitable times increase or decrease the number of processes they do have runnable. We have

tested the system with several applications. In many of the test cases the applications execute more than twice as quickly when our modified threads package is used.

A couple of notes on terminology. In this paper, the term *process* is used to refer to both lightweight and heavyweight processes that are preemptively scheduled by the kernel. We assume that multiple processes can share memory. For heavyweight processes this could be implemented with distinct address spaces that are overlapped through virtual memory (e.g., Unix System V and UMAX processes with explicitly shared memory), while for lightweight processes the shared memory is obtained by using the same address space (e.g., Mach kernel "threads" and V-System processes). The other term we frequently use in the paper is *tasks*. We consider the computation associated with a parallel application to be divided into small chunks, each of which may potentially execute in parallel. These chunks of parallel computation are called tasks. Tasks are assigned to processes in a coroutine-like manner for execution by user-level code, and over its lifetime, a single process may be responsible for the execution of several tasks. An example of this notion of tasks is user-level *threads* as implemented in the Brown University Threads package.

The rest of the paper is structured as follows. In Section 2, we describe the problem of process preemption in detail and discuss some of its effects on application performance. Section 3 describes related work and discusses why a new solution approach is needed. In Section 4, we discuss our scheme for avoiding process preemption by dynamically controlling the number of processes each application uses. Section 5 describes our implementation of this scheme in detail, and Section 6 presents performance data gathered from the implementation. Finally, in Sections 7 and 8, we discuss some of the directions we plan to explore in the future and present our conclusions.

2 The Problem Description

When several parallel applications are concurrently executing on a moderately-sized parallel machine, processes often must contend for processors. In order to fairly handle the excess number of processes, the processors are multiplexed between the processes, periodically preempting executing processes and scheduling waiting processes. A process is usually preempted due to the expiration of its time quantum, which occurs independently of the section of code the process is executing. Excessive preemption of processes can result in inefficiency from many sources, including time spent waiting for a suspended process to leave a critical section, time wasted in context switching, and inefficient cache behavior.

Figure 1 shows what can happen when a parallel application's processes must contend for processors. The data is gathered from an Encore Multimax with 16 processors. The graph shows the performance of two simultaneously executing parallel applications, a matrix multiplication and a one-dimensional FFT. Each application breaks its problem into a number of tasks, which are scheduled onto the processes executing that application. The figure shows the speed-up for the applications as the number of processes executing the tasks in each application is varied from 1 to 24. For example, we see that when both applications are started simultaneously with 24 processes each, the speed-up obtained by matrix multiplication is 2.8-fold and that for FFT is 2.4-fold.

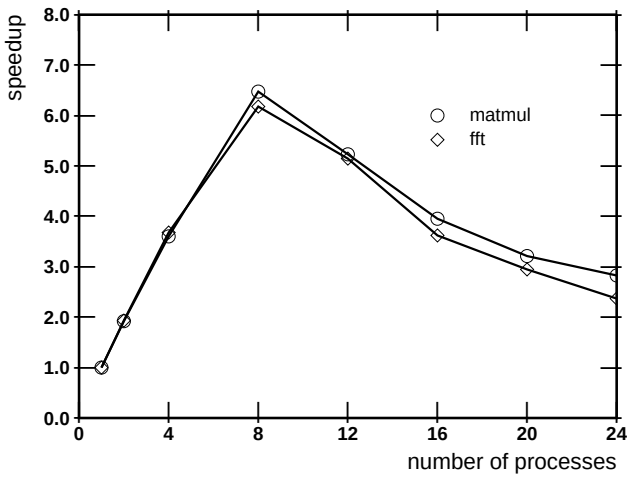

Figure 1: Speed-up when a matrix multiplication application and an FFT application are run simultaneously and the number of processes per application is varied.

Figure 1 shows that the performance of both applications worsens considerably when the number of processes in each application exceeds 8 and thus the total number of processes in the system exceeds the number of processors. Furthermore, the larger the number of processes the worse the performance gets, since it takes longer for preempted processes to be rescheduled. This is because unscheduled processes are placed on a FIFO queue, and the more unscheduled processes there are, the longer it takes for a preempted process to get to the front of the queue and be rescheduled. We also note that while an increasing number of running processes in a parallel application can often increase lock contention, this is not the reason for performance degradation seen in Figure 1. Since only scheduled, running, processes can contend for a lock, the contention when the number of processes exceeds the number of processors will be equivalent to the contention when the number of processes equals the number of processors.

The decreased performance in parallel applications when the number of processes in the system exceeds the available number of processors can be attributed to several problems. These problems include the following:

1. Processes may be preempted while inside spinlock-controlled critical sections. When this occurs, other processes executing the same application may end up waiting to enter that critical section, busy-waiting until the preempted process is scheduled for execution. While the chance of a process being preempted while inside a critical section is small if that critical section is small, the amount of time that may be spent waiting for the preempted process to be rescheduled is potentially very large. The problem is worst in fine-grained systems, where critical sections are entered frequently and are fairly large relative to the grain size.

2. Similar problems can arise between producer and consumer processes of an application. While a producer process is suspended, the consumer process may be scheduled to run on a processor only to realize that there is nothing for it

to do. Depending on how the synchronization between the producer and consumer processes is implemented, this can result in many wasted CPU cycles.

3. The frequent context switching that goes on when the number of processes greatly exceeds the number of processors is another source of performance degradation. Aside from the problem of corrupted caches (discussed below), a context-switch involves a number of system-specific operations (saving and restoring registers, switching address spaces, etc.) that do no real work.
4. Finally, a major source of performance degradation for high-performance multiprocessors is processor cache corruption. When a processor is interleaved between multiple applications, much of the data from previously scheduled applications will be purged from the cache on each time slice. By the time the old application is scheduled again, it will have to refetch its entire working set into the cache. Given that the cache miss penalty for remote accesses for some of the scalable multiprocessors currently being designed (e.g., the Encore Ultramax using the Motorola 88000 [17] and the Stanford multiprocessor using the MIPS R3000 [11]) will be 50-100 processor cycles, the performance degradation may be a factor of 10 or more.

In summary, serious performance degradation may occur when several parallel applications with a large number of processes are run on a multiprocessor. This will be especially true for applications which exploit parallelism at a fine granularity and for the new generation of scalable multiprocessors with very high cache-miss penalties. Consequently, one needs to be very careful in how application processes are scheduled onto processors, and efforts should be made to have the same process run on a processor for a long time.

3 Related Work

Since multiprocessors have become widely available only recently, not much research has been done in the past to address the issues raised in the previous section. We now briefly discuss some of the multiprocessor scheduling strategies others have proposed that attempt to address at least some of the problems involved.

The first one we consider is called *coscheduling*. It was proposed by Ousterhout [14] for the Medusa OS for the Cm* multiprocessor. In coscheduling, the multiprocessor scheduler is modified so that all runnable processes of an application are scheduled to run on the processors at the same time. Similarly, when an application is preempted, all of its processes are preempted, so processes will not be wasting processing resources busy-waiting. Effectively, the system context switches between applications, running all processes from one application for a time slice, then all processes from a second application for a time slice, and so on. While coscheduling handles the problems associated with spinlock-controlled critical sections and those with producer-consumer processes well (points 1 and 2 in the previous section), it does not reduce the overhead due to context switching or the degradation due to processor cache corruption (points 3 and 4). By the time an application is rescheduled on the system, the caches will probably not contain any of its data. In all fairness, though, we point out that processors in the Cm* machine did not have caches, so cache corruption was not an issue.

The second scheduling strategy that we consider is due to Zahorjan et al [18] from the University of Washington. They propose a smart scheduler for their threads package that avoids preempting processes when they are inside critical sections, and avoids rescheduling busy-waiting processes while a process inside a lock is suspended. In this approach when a process enters a critical section it sets a special flag, and the scheduler has been modified so that it will not preempt a process with this flag set. When the process leaves the critical section it resets the flag. While this scheme quite nicely prevents processes from being preempted inside a critical section, some of its weaknesses are the following. It has undesirable protection properties, in that it allows user processes to directly control the behavior of the scheduler. It makes incorrect decisions for some frequently occurring situations that arise in parallel applications. For example, it is common to have a hash table data structure with separate locks per bucket. It is reasonable in such applications to have many processes accessing different buckets in the hash table while holding the corresponding distinct locks. This scheme will prevent all of these processes from being preempted, even though they are not dependent on each other in any way. Finally, as in the case of coscheduling, this scheme does not address the issues of reducing context-switching overhead and reducing performance degradation due to processor cache corruption.

A third approach that combines facets of the previous two was proposed by Edler et al [7] at the NYU Ultracomputer project. In this proposal, processes can be formed into groups. The scheduling policy of a group of processes can be set so that either the processes are scheduled and preempted normally, or all processes in the same group are scheduled and preempted simultaneously (as in coscheduling), or processes in the group are never preempted. Also, an individual process can prevent its own preemption, independent of the scheduling policy of its group. This can be used to implement spinlock flags. While this approach is more flexible than the ones previously described, it also does not address the problems of cache corruption and context-switching overhead.

Finally, we consider some recent work from the University of Washington by Lazowska and Squillante [12] that evaluates the performance of several multiprocessor scheduling policies. The policies they consider rely on the "affinity" of a process for a processor, where the affinity is based on the contents of the processor's cache. The key idea is that a process should be scheduled on the processor on which it last executed (before being preempted), where hopefully a large fraction of its working set is still present in the processor's cache. However, if this policy is strictly followed it can lead to load imbalance, as processes are not allowed to migrate from busy processors to idle ones. Consequently, Lazowska and Squillante propose variations on the basic policy and evaluate the performance of the variations using queueing theory models. While the proposed approach helps in reducing cache corruption effects, it does not address the other problems that we discussed in Section 2. Furthermore, since the proposed policies have not been implemented yet, it is difficult to predict what the performance benefits will be in realistic situations.

4 The Proposed Solution

Unlike the solutions described in the previous section, our approach to the problem is based on the hypothesis that applications perform best when the number of runnable processes is the same as the number of processors. This hypothesis is suggested by experiments like the one shown in Figure 1, and is what we would like to validate in this paper. Intuitively, our approach is similar to that of a virtual memory machine, where the system works efficiently and avoids thrashing if the total number of pages in all running processes' working sets does not exceed the available physical memory.

The most important part of our solution is that applications should dynamically control their number of runnable processes to match the number of processors available to them. Once this is achieved, each runnable process will have its own processor. There will be no reason for preemption, the time spent context switching will be negligible, and there should be no degradation due to corruption of caches. Also, unlike coscheduling, spinlock flagging, and processor affinity schemes requiring kernel modifications, our method can be implemented entirely in user space. Of course, this description leaves many questions unanswered. Among them are how the number of processes in a running application can be varied, how the ideal number of runnable processes for each application should be determined, and how the real number of runnable processes should be made to match the ideal number. We divide the various issues into two categories, application-related issues and system-related issues, and consider them in that order.

4.1 Application-Related Issues

Before going into the mechanisms of process control, we need to discuss the question of when it is safe to limit the number of runnable processes in an application by suspending a process. By *safe*, we mean without potential starvation, loss of data, or significant loss of efficiency. A process can be safely suspended if and only if we can guarantee that either the results from the task it is executing will not be needed by the application until the process can be resumed, or that the task can be executed by some other process if necessary. For the general class of parallel applications, it can be difficult to determine when a process can be safely suspended.

Fortunately, for a slightly more limited class of parallel applications, the problem is much simpler. If an application can be broken up into a number of tasks (as defined in Section 1), where processes select tasks from a queue and execute them, then a process can be safely suspended after it has finished executing a task (or has put it back on the queue) and before it has selected another task to execute. As long as the application does not depend on having a certain number of processes executing, the number of processes can be varied without problems. We note that such task-queue based models are widely used to implement parallel applications on shared-memory architectures, so the restriction is not serious. For example, one can find several programming languages based on the task-queue model [3, 8, 10], and consequently all programs written in them follow the model. Similarly, most applications written using threads packages [6] follow this model, and also many independently written applications follow the task-queue model [9, 15, 16]. The task-queue model provides an easy way to achieve load balancing in shared-memory multiprocessors, and hence its popularity.

At this point in our implementation, we have only added process control support for applications written using threads packages. Our reason for doing so is that several operating systems provide libraries that implement threads for the application programmer. These packages give the programmer a few commands for creating and destroying threads and controlling access to data shared between threads, and hide the rest of the details from the application. One such library is the Brown University Threads package [6]. The implementation results we present in Section 5 are based on the Brown system. [1]

With applications written using threads packages, the application programmer breaks parts of his problem up into threads (which correspond to our "tasks"). When the application is started, some user specified number of processes are created to execute the threads. As the execution proceeds, the processes pick up threads for execution from a task queue. As the result of executing a thread of control, that thread may decide to add new threads to the task queue. The application ends when there are no more threads to execute. As an example of how process control would work using threads, consider an application that generates four threads. If four runnable processes are used to execute the code, each thread can be executed on a process at the same time. Now suppose the system load increases so that the application should be limited to three runnable processes. Then, if one of the threads blocks, the process executing it can be safely suspended after queueing the partially executed thread. After another thread finishes executing, the process that was running the completed thread can now dequeue and finish executing the suspended thread. Thus, the application programmer can write his code independently of the number of processes and processors that will be executing that code.

4.2 System-Related Issues

For our implementation, we divided the task of controlling application processes into two parts: (i) determining how many runnable processes each application should have, and (ii) restricting each application to the appropriate number. The functionality of each part could be put in the kernel, distributed among the applications, or placed in a user-level centralized server process. We now evaluate some of these options.

In order to determine the number of processes each application should have runnable, we need to know the number of processors in the system, the number of runnable processes used by applications that cannot be controlled, and the number of applications we are controlling. Implementing this functionality outside of the kernel in a centralized user-level server necessitates kernel system

[1] Another, similar, package is Mach's C-Threads package [5]. Under C-Threads, though, a new process is normally created whenever a thread is created, and killed when it finishes executing that thread. While C-Threads does have a mechanism for limiting the total number of processes used by an application, the paradigm of having a one-to-one correspondence between threads and processes does not work well with process control, where the number of processes must be independent of the number of tasks to be executed. For this reason, we did not use the C-Threads package for our implementation of process control.

calls to determine information about process status. However, distributing this functionality among the applications requires even more of these system calls, one for each application for each update interval. The kernel approach, on the other hand, requires extending the kernel.

For our current implementation, we have chosen to use the centralized user-level server approach. We experimented with the decentralized approach and found it to be too inefficient for our purposes. It also introduced stability problems that appeared to be only solvable with expensive communication protocols. Finally, in the short run, we did not want to modify the kernel, and the centralized server approach was found to be adequate. The results in Section 6 are based on this approach. In the long term, however, we think it will be necessary to modify the kernel. We discuss our ideas about kernel modifications and multiprocessor scheduling in Section 7.

To control the number of runnable processes in each application, the controller must be able to suspend and resume those processes. A user-level centralized server cannot do this, since it has no control over processes it has not spawned, aside from interprocess communication. That control must either come from the kernel or from the application itself. The problem with controlling the number of runnable processes from the kernel is that it is difficult for the kernel to determine when a process can be suspended safely; it needs information from the application to determine this. Letting the application suspend and resume processes is simpler and more efficient, and is the approach we have taken. We further note that since the scheduling and descheduling of threads from the task queue is managed by the threads package library rather than by the applications programmer, it is possible to implement the safe suspension and resumption of processes totally transparently to the user. We have actually done this, and the details are presented in the following section.

5 Implementation

In our current implementation we have chosen to use a centralized user-level server to help determine how many processes an application should have. The code for controlling the number of processes in an application has been embedded into a version of Brown University's Threads package, running on a 16 processor Encore Multimax with a variant of 4.2 BSD Unix called UMAX. The UMAX operating system provides interprocess communication through sockets, and also provides a system call for determining information about the runnable processes in the system. Both of these features were necessary to our implementation, though they could be replaced by similar features on other systems if we were to port our scheme to another machine.

The overall structure of our scheme is shown in Figure 2. It consists of a central server, and some number of processes partitioned in various ways. To determine the number of runnable processes that each controllable application should have, the server periodically calls the kernel and gets a list of all runnable processes. It first determines the number of runnable processes not belonging to controllable applications. It then subtracts this from the number of processors in the system, to determine the number of processors available for the use of the controllable applications in the system. It then partitions these processors among the applications

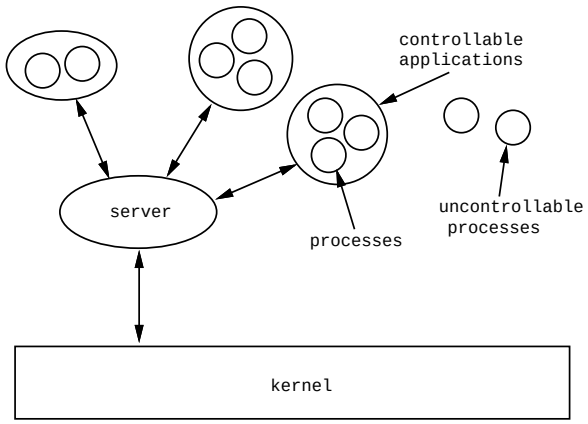

Figure 2: Basic structure of the server-based scheme.

fairly in order to determine the number of runnable processes each application should be using. Special provisions are made so that an application will not be "assigned" more processors than it can use; that is, the server makes sure that the number of runnable processes it thinks a given application should have does not exceed the total number of processes the application has. It also ensures that each application has at least one runnable process to avoid starvation.

For example, assume the system has the runnable processes and applications shown in Figure 2, and further assume the machine has 8 processors. The central server will determine that 2 processors are being used by uncontrollable applications, and proceed to distribute the other 6 among the three controllable applications. Given that all three have the same priority, each of them gets two processors. The first application with only 2 processes need not suspend any processes as it has 2 processors available, but the other two applications will have to suspend one process each to avoid context switching. As applications are created or completed, this process is repeated.

The direct control of the number of runnable processes of an application is performed by individual processes in that application. As each controllable application begins execution, the root process of the application sends a message to the central server notifying the server of the application's existence, and further telling it the process ID of the root process. This process ID is used to determine which processes belong to the application, by comparing it with each process' parent process ID. (It also indirectly helps the server determine the number of runnable processes not belonging to a controllable application.) The application then begins execution, and periodically a process in the application polls the server to determine the number of processes it should ideally have runnable. This polling is done every 6 seconds in the current implementation.[2] The application then receives and stores this information, and whenever a process in the application enters

[2]The interval of 6 seconds was chosen without much experimentation, and another choice might result in better performance. The 6 second interval was an attempt at balancing the overhead of process suspension and resumption against the overhead of having too few or too many runnable processes.

a section of code where it may safely suspend itself, it compares this number with the actual number of runnable processes in the system. If the ideal number is less than the actual number, the process suspends itself; if the ideal number is greater than the actual number, the process wakes up a previously suspended process. The suspension is done by having the process wait for a signal that will not ordinarily be generated by the system; the resumption is done by sending that signal to a waiting process, kept on a queue. If processes enter sections of code where they may be safely suspended fairly often, the actual number of runnable processes will approach the ideal number of runnable processes fairly rapidly, and will stay fairly close to it as circumstances change the ideal number.

The process control mechanisms in the threads package are actually hidden from the application programmer. The interface to the threads commands was not changed when process control was added. The process monitoring, suspension, and resumption, is done when the application returns control to the threads package when a thread is suspended or has finished execution. This lets almost all applications written using the threads package run using process control without any modifications whatsoever.

6 Results

In this section we present preliminary results from our implementation on the 16 processor Encore Multimax. The results are for several applications under a variety of system load conditions. All of the applications are implemented using the Brown University Threads package. The applications considered are the following:

`fft` A parallel single-dimension Fast Fourier Transform, based on an algorithm by Norton and Silberger at IBM Yorktown Heights [13]. This FFT algorithm has several loops that were broken into parts to provide parallelism.

`sort` A parallel merge sort algorithm, simultaneously sorting a number of small lists of numbers with heapsort, and then merging pairs of sorted lists in parallel until the final sorted list is achieved.

`gauss` A parallel Gaussian elimination algorithm. The solution is computed using partial pivoting and back substitution, and the row elimination is parallelized.

`matmul` A simple matrix multiplication algorithm. The multiplication is parallelized by splitting the multiplicand by rows.

Figure 3 shows the performance of the four applications described above. For each application we plot the speed-up as the number of parallel processes is increased. Two curves are shown for each application: (i) the dashed line shows the implementation of the application on top of the original, unmodified Brown Threads package, and (ii) the solid line corresponds to the implementation on top of our modified threads package that controls the number of processes. We would like to make the following observations. First, as expected, for each application the speed-up increases up to 16 processes, which is equal to the number of processors. Second, note that the dashed and the solid curves are almost identical up to 16 processes. This means that the overhead of our implementation is negligible, at least in cases where no reduction in the number of processes is necessary. Third, beyond

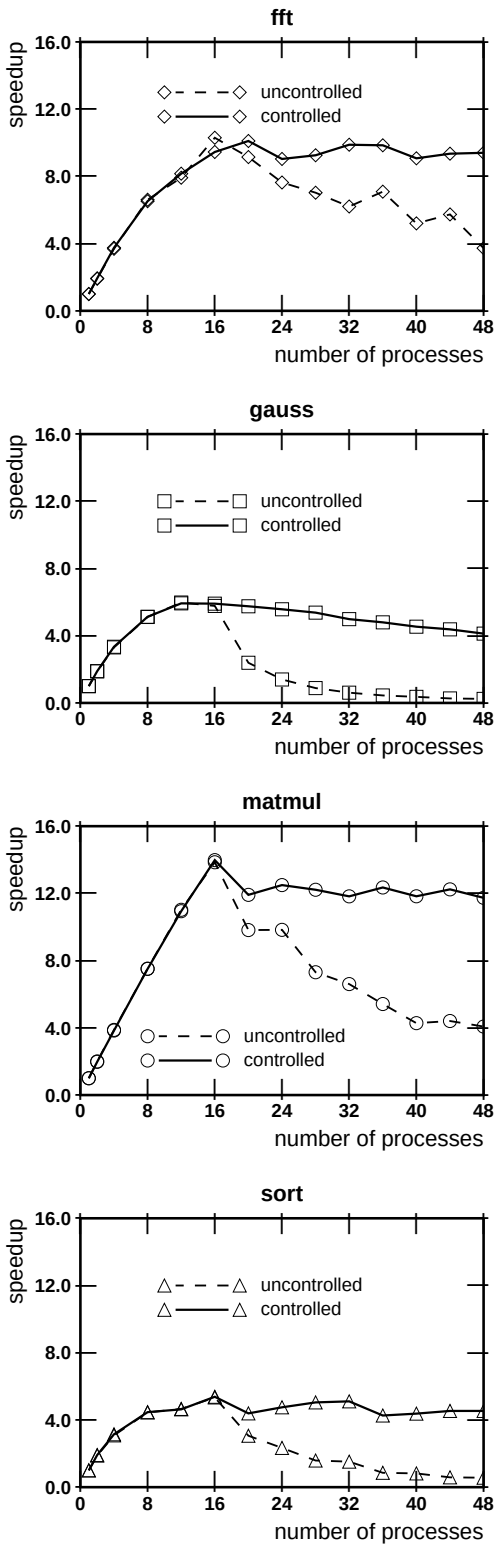

Figure 3: Comparison of speed-up as the number of processes is varied with and without process control in effect.

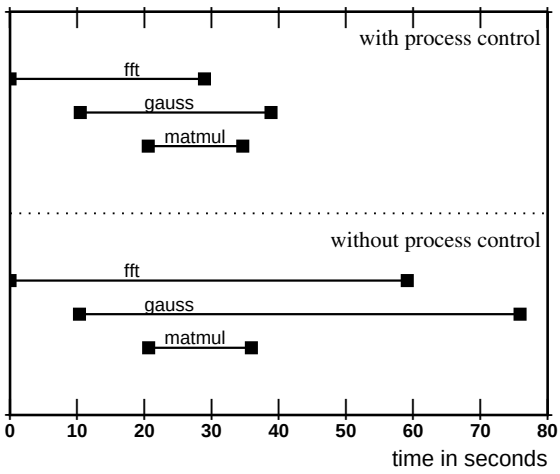

Figure 4: Execution profiles for `fft`, `gauss`, and `matmul` applications, each started with 16 processes, beginning execution at different times but overlapping with each other.

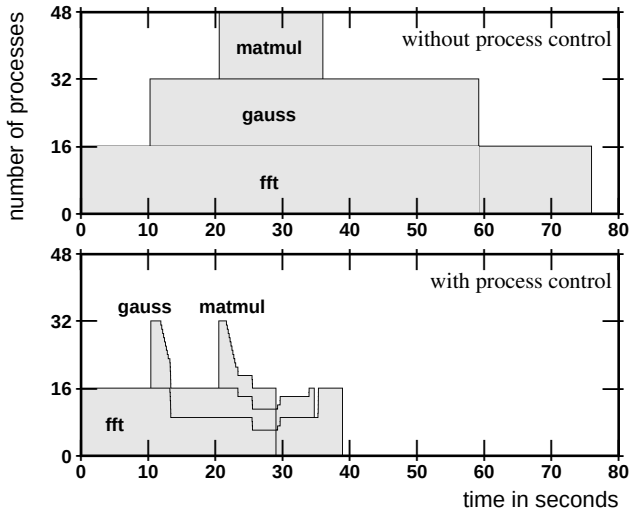

Figure 5: Total number of runnable processes in the system as a function of time. The top half of the figure corresponds to the case when number of processes is being controlled by our threads package. The bottom half shows the same run without process control.

the 16 process point, the speed-up with the unmodified threads package is significantly worse than that with ours. In fact, the larger the number of processes, the more the difference. The reason is that a process that is suspended within a critical section takes a longer time to get rescheduled if the number of runnable processes is larger.

While Figure 3 showed results when only one application was running on the system, a more realistic case is when there are several applications running concurrently. Figure 4 shows the results when three applications execute at the same time, both with and without process control. The applications were started at intervals of 10 seconds, each with 16 processes. As is clear from the figure, the wall clock time to execute the `fft` and `gauss` applications is much longer without process control than with it. By using process control, we are greatly increasing the performance of these applications. The time for the `matmul` is also increased, but not by a large amount. This is somewhat surprising because when `matmul` starts up under the uncontrolled version there are a total of 48 processes, and the performance for `matmul` under those conditions is pretty bad as shown in Figure 3. Our current guess is that it may be due to the structure of the Encore's scheduler, where processes just starting up may have higher priority than slightly older processes due to the relation of priority to past CPU use.

To provide a more detailed look at how process control is happening, we present the graphs in Figure 5. In this figure we plot the number of runnable processes in the system as a function of time. These plots correspond to the three application runs shown in Figure 4. We see that with process control turned on, the total number of processes quickly returns to 16, which is the number of processors in the system. The few seconds of delay before the number of processes starts decreasing is because applications query the central server only once every six seconds. We can also see how the processors are equally divided between `fft` and `gauss` during the time interval 10–20, and how they are divided among all three applications during the interval 25–30. Also note that the number of processes used by each running application starts increasing during the interval 30–40 as applications finish executing. Of course, with process control turned off, the number of processes increases much beyond the number of processors (see bottom half of Figure 5), and the performance suffers as a result. For example, the `gauss` application takes 66 seconds to execute instead of 28 seconds.

7 Future Work

Although the system described in this paper increases the efficiency of parallel applications without requiring changes to the kernels of typical operating systems, we think that in continuing this work we will need to make kernel modifications. This is particularly true if we are to effectively handle realistic multiprogramming environments where there will be a mixture of applications. There may be some parallel applications that control their processes, there may be others that don't, and there may be single-process applications like compilers, editors, and network daemons. One problem with using process control in such environments is that an application that does not control its processes may get an unfair share of the processors. The solution to this problem we have decided to implement involves partitioning the machine's processors fairly, and isolating the processes used by each application to a partition. A controlled application could then execute concurrently with an uncontrolled application, and still get a fair portion of the system processing resources. Another benefit is that processes running on any given processor would be from the same application, using many of the same code locations and data structures. This would help increase hit ratio of the caches and increase performance of individual processors.

We plan to implement the above by dynamically partitioning

processors in a machine into *processor groups*. As the overall system load and the requirements of individual applications change, the assignment of processes to processor groups and the number of processors in each processor group will vary. For example, there would usually be one processor group per parallel application. Similarly, there would normally be a separate processor group for single-process applications like compilers, OS daemons, etc. Of course, if the number of applications is larger than the number of processors, then multiple applications may have to be assigned to the same processor group. However, if applications that control their processes are kept in different processor groups from those applications that don't, then unfair hogging of processors can still be avoided.

We propose that the processor groups be managed by a high level *policy module*. The functions of the policy module would be to decide: (i) when to create or delete processor groups; (ii) how to distribute the processors among the processor groups; and (iii) how to assign applications to processor groups. We envision scheduling of processes within a processor group happening at a lower level. We expect that each processor group will have its own separate run queue, and will pick processes from that queue using standard priority mechanisms. Splitting the functionality between a central policy module and distributed scheduling will hopefully prevent the policy module from becoming a bottleneck.

8 Conclusions

In this paper, we considered the problem of performance degradation of parallel applications when the number of runnable processes greatly exceeds the number of available processors. This is a frequent occurrence in multiprogrammed parallel systems. In these situations, processor time may be wasted due to busy-waiting trying to obtain locks held by suspended processes, waiting for data from suspended producer processes, context-switching overhead, and processor cache corruption. These problems, particularly the last, will be even more significant on the scalable high-performance multiprocessors currently being developed. We propose that by dynamically controlling the number of processes each application uses, we can keep the number of runnable processes in the system close to the number of available processors, thus avoiding process preemption and the associated costs.

We have implemented a system for controlling the number of runnable processes used by applications, and have compared the performance of applications both with and without process control when executing on a 16-processor Encore Multimax. The applications executed much faster with process control than without. While our current implementation runs only in conjunction with Brown University's Threads package, it should be easily portable to other systems providing software thread support. An important feature of our implementation is that it requires no modifications to the user application code. Only the underlying threads package needs to be modified. Our scheme should also increase the efficiency of task-queue based parallel programming languages like QLISP if used in their run-time systems.

Our current implementation has certain limitations when applications that control their processes are run with applications that do not control their processes. In such cases, the applications without control end up with a disproportionate amount of processor time. To avoid this, we plan to modify the process scheduler in such a way that processors are partitioned fairly, and each partition executes only the processes of one application. This should avoid the problem of greedy uncontrolled applications, and should also help increase cache hit ratios. We are currently incorporating our ideas into the scheduler associated with the V distributed operating system [4].

Acknowledgments

We would like to thank David Cheriton, Hugh Lauer, Edward Lazowska, and Susan Owicki for their helpful comments on this paper and the work it describes. This research was sponsored by DARPA contract N00014–87–K–0828.

References

[1] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

[2] BBN Laboratories Inc. *Butterfly Parallel Processor Overview*. 1986. BBN Report No. 6148.

[3] Rohit Chandra, Anoop Gupta, and John Hennessy. COOL: A language for parallel programming. In *Proceedings of the Second Workshop on Programming Languages and Compilers for Parallel Computing*, University of Illinois, August 1989.

[4] D.R. Cheriton. The V distributed operating system. *Communications of the ACM*, 31(2):105–115, February 1988.

[5] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, 1988.

[6] Thomas W. Doeppner, Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[7] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. Technical Report Ultracomputer Note 136, New York University, 1988.

[8] Richard P. Gabriel and John McCarthy. Queue-based multiprocessing Lisp. In *ACM Symposium on Lisp and Functional Programming*, pages 25–43, 1984.

[9] Anoop Gupta, Charles Forgy, Dirk Kalp, Allen Newell, and Milind Tambe. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[10] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] John Hennessy et al. DASH: A directory-based scalable general-purpose shared-memory multiprocessor. In preparation, Computer Systems Laboratory, Stanford University, August 1989.

[12] Edward D. Lazowska and Mark S. Squillante. Using processor-cache affinity in shared-memory multiprocessor scheduling. Department of Computer Science, University of Washington, Seattle, June 1989.

[13] Alan Norton and Allan Silberger. Parallelization and performance prediction of the Cooley-Tukey FFT algorithm for shared-memory architectures. Technical Report RC 11237, IBM Research Division, 1985.

[14] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[15] Jonathan Rose. LocusRoute: A parallel global router for standard cells. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, June 1988.

[16] Larry Soule and Anoop Gupta. Characterization of parallelism and deadlocks in distributed digital logic simulation. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, June 1989.

[17] Andrew W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 244–251, 1987.

[18] John Zahorjan, Edward D. Lazowska, and Derek L. Eagar. Spinning versus blocking in parallel systems with uncertainty. Technical Report 88-03-01, Department of Computer Science, University of Washington, 1988.