オペレーティング・システム　56－10
(1992. 8. 19)

対称型マルチプロセッサにおける2階層スケジューリング方式

Joe Uemura　菅　隆志
三菱電機（株）情報電子研究所
email: {uemura,kan}@isl.melco.co.jp

概 要

従来の対称型マルチプロセッサでは、各プログラムの並行実行によりシステム全体のスループットを向上できるものの、その環境下で特定プログラムの並列処理を効率よく行ないその処理性能を保証することが困難であった。この問題を解決するため、筆者らは、オペレーティングシステムとユーザモジュールの2階層からなるスケジューリング方式を提案し、タイムシェアリング環境下においても各ユーザプログラムの並列処理を保証し、かつシステムスループットへの影響をおさえることができることを示す。本論文では、この実現方式と並列処理形態について述べると共に、実際の対称型マルチプロセッサへの適応結果を報告する。

A Two-level Processor Scheduling Scheme for Shared-memory Multiprocessors

Joe Uemura and Takashi Kan

Mitsubishi Electric Corporation
Computer & Information Systems Laboratory
5-1-1 Ofuna, Kamakura, Kanagawa 247, Japan
email: {uemura,kan}@isl.melco.co.jp

Abstract

Traditional scheduling techniques used in timesharing operating systems usually perform inefficiently when applied to parallel programs. In this paper, we propose a two level scheduling scheme for shared-memory multiprocessors where scheduling decisions are made by an user module and the operating system. The scheme provides for guaranteed performance of parallel codes while accommodating the execution of timesharing applications at the same time. This paper proposes both an implementation and a parallel execution model. Our preliminary results show that, using this scheme, sustained performance of parallel applications can be obtained regardless of other concurrently executing applications.

1. Introduction

A large number of shared-memory multiprocessor systems are being offered commercially. These systems are often utilized for general purpose uses; the multiple processors produce higher throughput by running several sequential applications simultaneously. The availability of multiple processors, however, offer the additional benefit of speeding up a single application by using parallel processing. In parallel processing, different components of an application execute on several threads. Each of these threads of execution can be assigned to a different processor, leading to shorter application execution times.

In this paper, we are mostly concerned with microprocessor based shared-memory multiprocessor systems. Most of these systems run some variation of UNIX[1] adapted to support multiple processors. The schedulers used in these systems are often modified versions of the UNIX timesharing scheduler. The timesharing scheduling policies in these schedulers frequently fail to satisfy the needs of parallel applications, leading to severe performance degradation of parallel codes. Particularly, when the number of running threads exceed the number of available physical processors in the system, the timesharing policies introduce system behavior inappropriate to the needs of parallel applications. This behavior is characterized by the schedulers interfering with how resources are allocated to applications. As threads contend for processor resources, the system will multiplex the physical processors over the threads according to their own notion of the importance, ie. priority, of threads. This is accomplished by preempting running threads and scheduling waiting ones. Some of the reasons why these policies are inappropriate when applied to parallel applications are:

• lack of control over resource allocation

In order to enjoy the benefits of higher application performance offered by parallel processing, applications have to be parallelized to exploit the multiprocessors. Parallelizing an application is usually a difficult task; without guaranted performance benefits, few would spend the extra effort. Providing a predictable environment where guaranteed performance can be obtained is highly desirable when running parallel applications. If the operating system relies on its own criteria to allocate processor resources, a parallel application might not receive the necessary resources, leading to unacceptable performance degradation.

• parallel applications synchronization patterns

Multiple threads in parallel applications often have to synchronize during execution. Common synchronization schemes use a busy-wait strategy. If a thread is preempted while holding a busy-wait synchronization lock, the thread(s) waiting on the lock will loop until the thread holding the lock is rescheduled and releases the lock. Another common synchronization mechanism is the barrier construct. A barrier synchronizes a predetermined number of threads ensuring that those threads have completed a certain portion of code before proceeding to the next. For efficient barrier execution, the threads synchronizing at the barrier should execute at similar rates. Preemption of threads might cause an uneven pattern of execution making the synchronization at the barrier very inefficient.

• undesired overhead due to context switches

Preemption of threads during execution causes undesired overhead to be added to execution time, due to the costs of context switching. Also, as threads migrate between the different processors during context switches, cache corruption can occur. This also adds to execution times, especially with shared-memory single-bus machines where the increased number of accesses to memory due to cache misses leads to higher bus contention.

Currently, when parallel applications run on

[1]Developed and Licensed by UNIX System Laboratories, Inc.

these types of multiprocessors, they usually run on dedicated machines. The operating system is brought up in single-user mode, and the entire system is dedicated to execute parallel applications one at a time in batch mode. This offers the desired predictable performance environment at the expense of having to dedicate the entire system to run parallel applications. This approach avoids the problem of other loads interfering with the execution of parallel codes by simply stopping all timesharing activities.

As a possible solution to these problems, we propose a two-level scheme which allows the system to provide the necessary resources to parallel applications, while accommodating timesharing applications at the same time. Our approach allows applications to participate together with the operating system when scheduling decisions are made. The scheme proposes both an implementation and a parallel execution model for fine-grained parallel programming. We have implemented an experimental version of this scheme on top of the MACH operating system running on a shared memory multiprocessor.

The rest of this paper is organized as follows. Section 2 introduces our proposed two-level scheduling scheme, its structure and implementation. Section 3 describes a parallel execution model which can be used by parallel applications that want to use our scheme for obtaining sustained performance regardless of the system load. Section 4 presents preliminary results gathered from the implementation, while section 5 discusses related work. Finally, section 6 offers our conclusions.

2. A two-level scheduling scheme

2.1. Processor Partition

Our approach employs a processor partition mechanism [Gupta91]. This mechanism permits the system to be divided into any number of partitions. A partition is a set of processors which have been combined together. Each partition is essentially a separate computing resource which can be adapted to meet specific application requirements. An application which wants to control its processor resources creates a partition where its tasks and threads will run. The application also requests that a number of processors be allocated to the partition. These processors are then scheduled only within the partition, making a controlled environment suitable to the specific needs of the application. Figure 1 illustrates the relationship between partitions, tasks, threads, and processors.

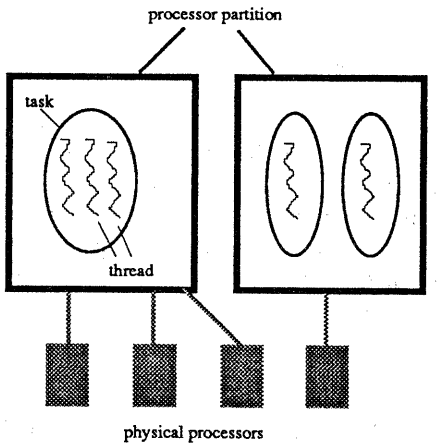

Figure 1.

Relationship between processor partition, physical processors, tasks, and threads.

When the system is initialized, a default partition containing all processors is created. All tasks and threads, including system daemons, and timesharing applications run in this default partition. This default partition is guaranteed to have at least one processor assigned to it at all times. This assures that applications which do not explicitly control their processor requirements will not suffer processor starvation due to lack of processor resources.

We choose MACH as the base operating system to implement our processor partition strategy. MACH is a distributed multiprocessor operating

system developed at Carnegie Mellon University. The MACH kernel itself only implements basic services, processor scheduling being one of them. The MACH processor scheduler has provisions for both timesharing scheduling and extensions to support processor allocation. These extensions are the basic mechanisms to implement processor partitions. In MACH terminology, the kernel supports processor sets and processor objects. Processor partitions are implemented using these objects. For a more detailed explanation of the MACH scheduler, and its processor allocation extensions, refer to [Black90a].

2.2. Structure and Implementation

Our scheme implements a two-level scheduler. In this class of schedulers [Gupta91], the high level component implements the policies which determine how processors are partitioned, or divided, among different applications. Scheduling of threads within a partition is done by the low level scheduling component in the operating system.

Figure 2 illustrates the components of the scheme. The MACH kernel provides the basic mechanisms for processor allocation. At the high level, a central server implements the policies concerning how processors are allocated to competing partitions. This server manages a pool of available processors, and implements the allocation policies by satisfying requests from applications which want to control their processor requirements. Applications interface with the server via a runtime library.

Our approach implements a *semi-dynamic* allocation scheme. Traditionally, processor allocation schemes are static or dynamic [Zahorjan90]. In static allocators, the number of processors allocated to an application remains constant during the entire application execution. By contrast, in dynamic schemes, the number of processors may actively vary during the application execution. Thus, dynamic allocators can better adjust to the changes in the system loads, although the cost incurred in reallocating processors can nullify its advantages over static schemes. To avoid this cost, our implementation is basically static; however, an application using our scheme can cooperate by letting the processor allocator in the server know that the application cannot make full use of its allocated resources. At those times, and only at those times, the system might choose to reallocate processors if the system conditions so demand. This application participation is an important aspect of our implementation. Unlike fully dynamic schemes where the system alone makes the decisions, we chose to have the application participate because only the application understands its processor resource requirements as they vary over time, and can evaluate the performance tradeoffs.

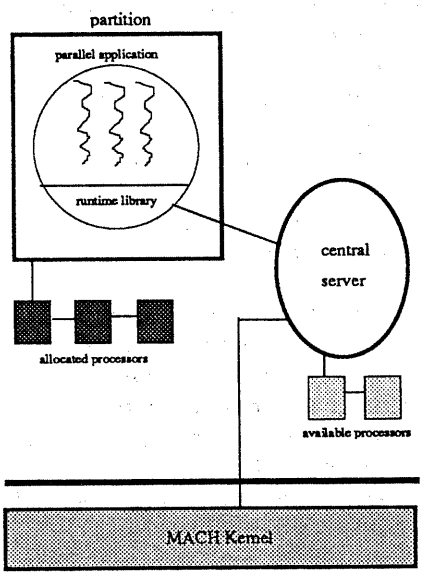

Figure 2.

Components of our scheme

The central server implements the high level scheduling policy by allocating processors to applications while the kernel provides the low level scheduling.

We now describe the interactions between the

components of our scheme. Applications which want to control their processor resources invoke functions in the runtime library. The runtime library usually makes requests for processors at the beginning of execution. Once applications no longer need the processor resources, they again call the runtime library which invokes the server to release the processors. The server will then either place these processors back to a pool of available processors, or use them to satisfy pending requests.

An application may choose to tell the server that it can not make full use of its allocated processors. This gives a hint to the server that this would be an appropriate time to take processors away from this partition. The server then may choose to assign the processors to a different partition. It is important to note that this is simply a hint; if there is no pending request for processors by another partition, the server will not take processors away, saving the cost of reallocating processors.

3. Parallel Execution Model

In this section, we describe our parallel execution model. This execution model can be used by fine grained numerical parallel applications that want to control their processor resources. Our work focuses on providing a construct which allows efficient loop parallelization. We chose to concentrate on efficient loop execution because numerical programs spend most of their time executing loops.

Our execution model is based on microthreads, also commonly known as microtasks [Cray] [Sequent]. Our implementation of microthreads works as follows:

1) the initial thread in the program creates a number of helper threads, or microthreads. These threads remain spinning, waiting for the main thread to tell them to start executing. The main thread receives a handler which is used to control the microthreads.
2) the initial thread then executes sequential code, if any is available. When a parallel block[2], usually containing a loop of some form, is reached, the main thread sets up the block for parallel execution, and tell the microthreads via the handler to go execute the block.
3) all microthreads, together with the main thread concurrently execute the parallel block. All microthreads synchronize at the end of the block via a barrier mechanism.
4) once all microthreads have synchronized, the call returns, and the microthreads will be spinning again, waiting for the next parallel block.

Our execution model introduces the concepts of *free* and *bound* microthreads. *Free* microthreads run in the default partition, without any processor allocation control. *Bound* microthreads always run on a processor partition which the runtime environment creates on behalf of the application. In the current implementation, the number of *bound* microthreads always matches the number of physical processors assigned to the partition. This provides sustained performance by guaranteeing the allocation of necessary processor resources.

This parallel execution model, although simple, can be very effective for numerical parallel programs which deal with large arrays of data. The parallelism inherent in the data allows efficient execution of multiple threads.

4. Performance results

In order to evaluate the effectiveness of our scheme, we executed two parallel applications: a matrix multiplication and a 2-dimensional FFT. These applications were run under two environments. One environment used *bound* microthreads while the other, *free* microthreads.

The results are shown in figure 3. To evaluate the affect of background loads on the the

[2] a block of statements which are executed in parallel

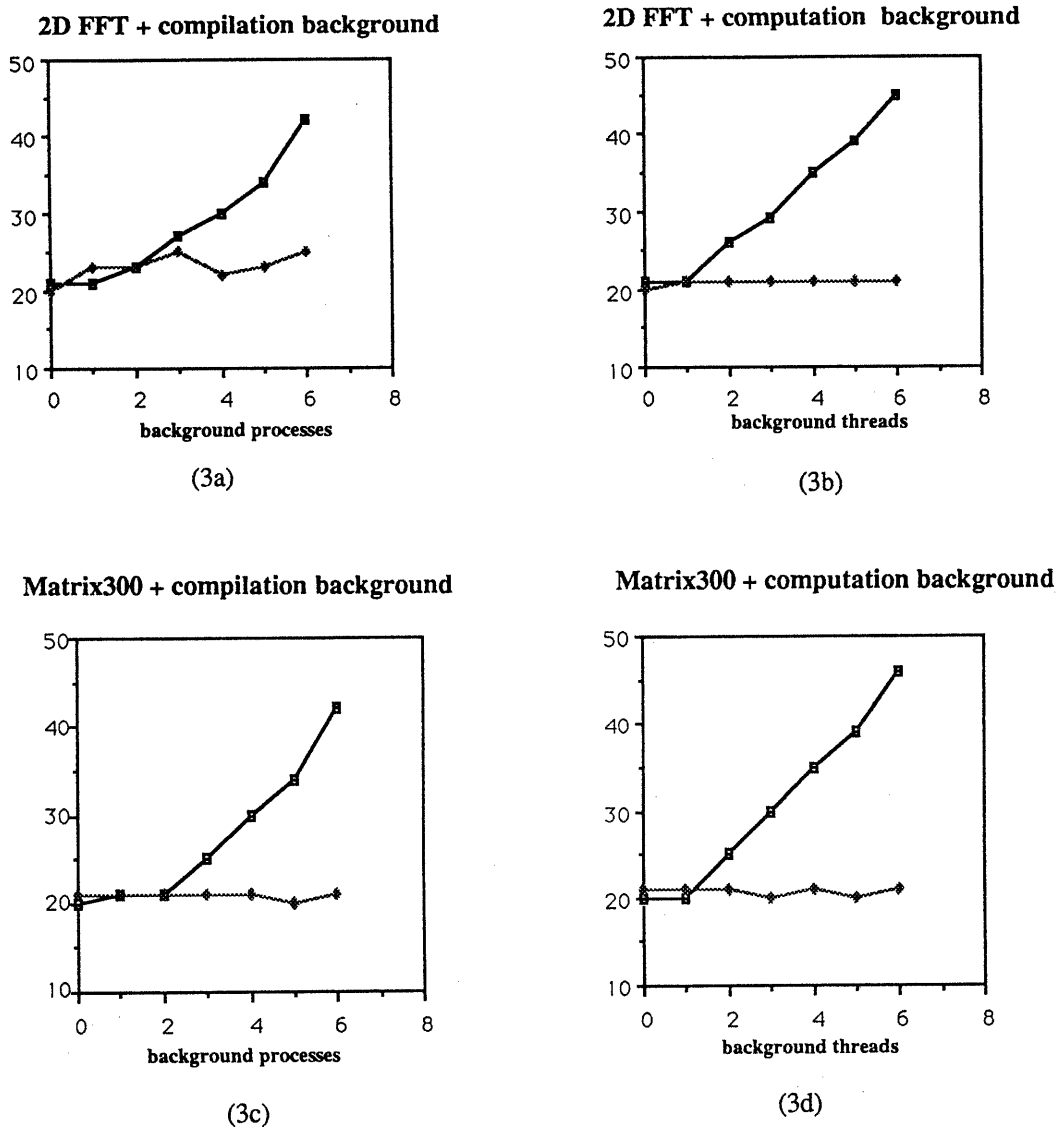

Figure 3.

Execution times in seconds are given in the vertical axis of each plot. The measured parallel applications are a 64x1024 2D FFT and a 300x300 matrix multiplication. Concurrently executing with the measured parallel applications are two types of background loads: compilation and computation. The horizontal axis in the compilation plots (3a and 3c) represent the number of concurrrently executing compilation processes. In the computation plots (3b and 3d), the horizontal axis indicate the number of threads executing in the parallel applicaton being run in the background.

performance of the measured parallel applications, we generated two types of loads:

1) *compilation load*: this is an attempt to introduce timesharing background loads by running compilations, a common activity on systems used for software development.
2) *computation load*: this load type illustrates the system behavior when other computational intensive parallel applications are run at the same time as the measured parallel application.

Our measurements were taken on our four processor shared-memory prototype system. The measured parallel applications always ran on three microthreads. The results show:

- as the number of processes or threads in the background loads saturate the system resources, the parallel application running with *free* microthreads displays severe performance degradation.
- the measured parallel application using our scheme was not affected by the increase system load, delivering sustained performance with little degradation.
- computation intensive parallel loads had a more severe impact than timesharing loads on the performance of application running with *free* microthreads.

Overall, parallel applications running with *free* microthreads, ie. without a processor partition, suffered severe performance degradation. Execution times increased by a factor of two or more when the number of background processes or threads was six.

5. Related work

Several researchers have suggested using two-level schedulers for shared memory multiprocessors. Zahorjan & McCann in [Zahorjan90] give an extensive comparison of static and dynamic processor allocation approaches. Although their work suggests possible implementation schemes, their evaluation is based on simulation, and actual real system implementations are not presented. Tucker & Gupta in [Tucker89] propose a process control scheme for multiprogrammed shared memory multiprocessors. Their approach is similar to ours; the structure of the components follows the same pattern of using a centralized server which implements an allocation policy. However, their implementation was done entirely in user-level without kernel support to perform processor allocation. This results in the possibility of uncontrolled processes using more resources than what is assumed by their process control scheme. This is one of the problems which our scheme wants to avoid: varying timesharing loads interfering with the performance of parallel programs. Their description of future work points that out, and they agree that kernel support is necessary to enforce processor allocation.

Using a global server to implement processor allocation on top of the MACH kernel has been implemented by Black [Black90b]. Unlike his implementation, our server uses prioritized requests and can allocate an optimum number of processors without clients having to specify the number of processors.

Microtasking, or microthreading has also been proposed by, among others such as [Cray], [Sequent] and [Doeppner87]. Our implementation and interface closely resemble Doeppner's Brown Threads microthreads. However, the difference in our implementation is that our microthreads can be explicitly bound to processors.

6. Conclusion and Future work

A large number of microprocessor based shared-memory multiprocessors are available. Taking advantage of the multiple processors to achieve faster execution times requires parallellizing the applications. A lot of the work done by parallel programmers is experimental, where the programmers want to measure the effectiveness of their parallelization techniques.

Supporting this type of activity requires an environment which delivers consistent performance, despite of varying system loads. The scheduling policies in these shared-memory multiprocessors often fail to support this type of environment to parallel programming. Instead of relying on running these applications on dedicated systems, we proposed a two-level scheduling scheme. This scheme offers a controlled environment to parallel applications while accommodating timesharing loads at the same time. Our performance measurements gathered on this initial implementation show that, without this processor control scheme, the performance can dramatically degrade as the system load increases. We have measured the affects of both computational and timesharing types of loads. As expected, the performance degradation is higher when computational loads are executed together with parallel applications.

Our work proposes a *semi-dynamic* processor allocation scheme. This makes the application controlling its resources responsible for cooperating with the system. There are still some hard questions regarding this scheme when looking from the timesharing side of the system. We need to study the effects of this policy on the total system throughput. We also would like to conduct some profiling, and perhaps devise a set of tools which would automatically place calls to allocate and release processors. These tools would take into consideration the profiled data, and the cost to allocate and release processors.

Acknowledgments

We would like to thank Vu Le Phan for his assistance and valuable comments on the presentation of this paper.

References

[Black90a] David L. Black. Scheduling and Resource Management Techniques for Multiprocessors. *Technical Report CMU-CS-90-152*, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Black90b] David L. Black. Scheduling Support for Concurrency and Parallelism in the MACH Operating System. *IEEE Computer*, 23(5), pages 35-43, May 1990.

[Cray] Cray Technical Manual. *CRAY Y-MP and CRAY X-MP Multitasking Programmer's Manual - SR0222E*. Cray Research Inc., 1988.

[Doeppner87] Thomas W. Doeppner, Jr. Threads: A System for the Support of Concurrent Programming. *Technical Report CS-87-11*, Brown University, June 1987.

[Gupta91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. *In Proceedings of SIGMETRICS '91*, pages 120-132, 1991.

[Sequent] Sequent Technical Publications. *Guide to Parallel Programming on Sequent Computer Systems*. Prentice Hall, New Jersey, 1985.

[Tucker89] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. *In Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159-166, 1989.

[Zahorjan90] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. *In Proceedings of SIGMETRICS '90*, pages 214-225, 1990.