Regular Paper

A Message-Pool-Based Parallel Operating System for the Kyushu University Reconfigurable Parallel Processor —Parallel Creation of Multiple Threads—

KUNIHIKO TSUNEDOMI*, AKIRA FUKUDA**, KAZUAKI MURAKAMI*** and SHINJI TOMITA****

The Kyushu University Reconfigurable Parallel Processor system under development is a MIMD-type multiprocessor which consists of 128 processing elements, interconnected by a full (128 × 128) crossbar network. Reconfigurable memory architecture employed by the system allows the system to be configured as either a shared-memory TCMP (tightly coupled multiprocessor), a message-passing LCMP (loosely coupled multiprocessor), or a hybrid of the two.

A parallel operating system under development is for the shared-memory TCMP, and aims at extracting various kinds of parallelism of the operating system itself to provide high-performance. To exploit the parallelism, the operating system is constructed by using a message-pool mechanism. A typical example of the parallelism is the parallel creation of multiple threads. In this paper, we propose four schemes for the parallel creation; the simple parallel scheme, the parallel template scheme, the chunk scheme, and the combination scheme.

Simulation results show that the chunk scheme is the most desirable among the schemes.

1. Introduction

The Kyushu University Reconfigurable Parallel Processor system under development is a MIMD-type multiprocessor which consists of 128 processing elements, interconnected by a full (128 × 128) crossbar network [1-3].

The goal of the system can be summarized as: i) to construct a high-performance, multi-purpose multiprocessor system which can be tailored to a broad range of applications, and ii) to offer an experimental environment, or testbed, which encourages many researchers in the studies of highly parallel processing.

The system employs reconfigurable network and memory architectures. The reconfigurable network architecture allows network topologies to match communication patterns of parallel algorithms. The reconfigurable memory architecture also allows the system to be configured as either a shared-memory TCMP (tightly coupled multiprocessor), a message-passing LCMP (loosely coupled multiprocessor), or a hybrid of the two. In the shared-memory TCMP, the memory architecture of the system supports uniform-memory-access (UMA) and nonuniform-memory-access (NUMA) architectures.

The operating system should reflect the flexibility and reconfigurability provided by the hardware system. However, we take a step-by-step approach to build the operating systems [4]. An operating system of the first step is for the shared-memory TCMP, because:

- The system configured as the shared-memory TCMP has a potential high-performance. We believe that exploiting an operating system which can extract the potential gain provided by the shared-memory TCMP contributes to research on highly parallel processing including operating systems.
- Exploiting such an operating system would also give us some quantitative and qualitative information on effective utilization of the flexibility and reconfigurability provided by the hardware system.

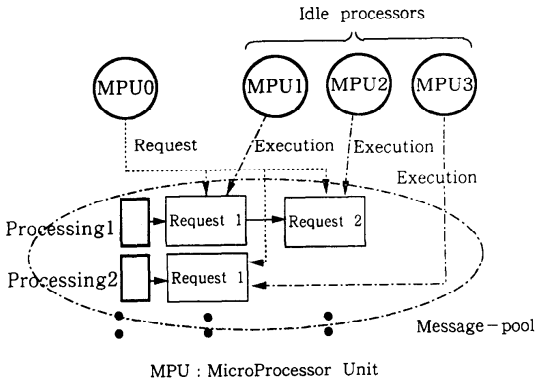
Some operating systems have been constructed or are under construction for the shared-memory TCMP [5-8].

*Hitachi Research Laboratory, Hitachi Ltd., 4026 Kuji-cho, Hitachi-shi, Ibaraki 319-12, Japan.
**Department of Computer Science and Communication Engineering, Faculty of Engineering, Kyushu University, 6-10-1 Higashi-ku, Fukuoka-shi, Fukuoka 812, Japan.
***Department of Information Systems, Interdisciplinary Graduate School of Engineering Sciences, Kyushu University, 6-1 Kasuga-Koen, Kasuga-shi, Fukuoka 816, Japan.
****Department of Information Science, Faculty of Engineering, Kyoto University, Hon-machi, Yoshida, Sakyo-ku, Kyoto 606, Japan.

Some of these are not interested in extracting the parallelisms of the operating systems, or extract system-call-level/function-level parallelism.

The operating system under development aims at extracting various kinds of parallelism of the operating system itself to provide high-performance. To exploit the parallelism, the operating system is constructed by using a message-pool mechanism. A typical example of the parallelism is parallel creation of multiple threads, which allows multiple threads to be created in parallel. Performance of the parallel creation depends on tradeoffs between enhanced performance due to parallel processing and overhead due to message handling and network/memory contentions.

This paper describes the performance evaluation of the parallel creation of multiple threads.



MPU : MicroProcessor Unit

Fig. 1 Parallel processing in kernel based on message-pool mechanism.

2. Design Principles and Message-Pool-Based Operating System

2.1 Design Principles

The design principles of the operating system for the shared-memory TCMP are as follows:

1) Provide the users with a well-matched parallel processing model to the shared-memory TCMP.

The operating system provides a process-thread model to the users. A process consists of an execution environment and multiple control flows (i.e. threads). The execution environment includes a paged virtual address space and protected access to system resources. A thread is the basic unit of CPU utilization (i.e. a control flow) [9]. It contains a context; a program counter and the contents of registers. We believe that the process-thread model is well matched to TCMP architecture, because all threads in a process share its virtual address space. The threads in a process work cooperatively. Therefore, considering the model, we should choose scheduling and memory management schemes.

2) Exploiting parallelism in a kernel.

To provide a high-performance kernel, we employ the following kernel processing style:

- The code of executing the system call is partitioned into various functions [10,11]. The functions are executed concurrently or in parallel, if possible. We call this parallelization type first-class of parallelization.
- The processing of a system call is organized as multiple control flows, each of which has the same code but different data from each other, if possible. The control flows are executed concurrently or in parallel. Of course, it depends on the type of system calls. We call this parallelization type second-class of parallelization.

2.2 Message-Pool-Based Operating System

A scheme which realizes both the first- and second-classes of parallelization is implemented by using message-pool mechanism in the kernel as shown in Fig. 1. When the kernel on a processor (e.g., MPU0 in Fig. 1) gets work which can be parallelized, the kernel puts request messages into the corresponding queues for idle processors to handle the requests in parallel. When the kernel gets work of the first-class of parallelization, the kernel partitions the work into subwork each of which has a different function and puts the messages for the subwork into the corresponding queues. In the case of the second-class of parallelization, the kernel puts the message or messages into the corresponding single queue. A typical example of the second-class of parallelization is the case where a multiple-thread-creation system call is issued. The message-pool mechanism allows multiple threads to be created in parallel and quickly. Performance of the scheme depends on tradeoffs between enhanced performance due to parallel processing and overhead due to message handling and network/memory contentions. We pursue the viability of the scheme.

3. Parallel Creation of Multiple Threads

3.1 Thread Creation

The operating system provides the multiple-thread-creation system call. The kernel has thread control blocks (THCBs) associated with threads. It contains context of the corresponding thread such as a stack pointer, a program counter, contents of CPU registers, thread state, and so on. To create a thread, the kernel normally performs the following processing:

1) Examining a free-THCBs-list, and picking up a free THCB.

2) Setting the appropriate information in it.

3) Allocating a stack for the thread.

4) Queueing the THCB into the corresponding thread ready queue, where the thread ready queue is associated with the process; all threads in a process share the same thread ready queue.

In the step 2), the information consists of two types of information:

•Private information: Each thread has different information from another. The information of this type includes a thread identifier, a stack pointer, thread state, and so on. A part of the information is produced by modifying the current information, which are shared by all processors. For example, the thread identifier would be provided by incrementing the current thread identifier.

•Common information: The information of this type is common to all threads created by the multiple-thread-creation system call. The information is used when the threads are about to run. The information includes arguments passed between the parent thread and the daughter threads. The information is copied from the stack of the parent thread to the stacks of the daughter threads through the kernel stack.

If the system call is sequentially processed in the kernel, its execution time is considerable when the number of threads to be created is great. Even if there are idle processors, the system call would be processed by a single processor. To reduce its processing time, we exploit parallelism at this level. The message-pool mechanism allows multiple threads to be created in parallel by idle processors.

3.2 Simple Parallel Scheme

One of the most intuitional schemes for parallel creation of multiple threads is that each idle processor creates a thread at a time by using the message-pool. Figure 2 shows the flow diagram of the simple parallel scheme. When the multiple-thread-creation system call is issued on a processor, the kernel on the processor constructs a message and puts the message into the corresponding message queue in the message-pool for idle processors to handle it. The message contains the number of threads to be created. An idle processor examines the corresponding message queue and creates a thread each time of the message access until all threads are created.

3.3 Issues on the Simple Parallel Scheme

Performance of the simple parallel scheme depends on tradeoffs between enhanced performance due to the parallel processing and overhead due to the message handling and memory/network contentions. The overhead includes the following:

(1) Overhead due to message handling

Overhead due to message handling occurs:

1) A message must be created to enque the corresponding message queue. This message construction is performed only when the system call is issued.

2) Idle processors access the message queue to read the message. The access occurs whenever the processor creates the thread.

(2) Serial bottlenecks

The parallel creation may produce some bottlenecks. There are two types of bottlenecks:

1) Shared data-structures access bottleneck: Shared

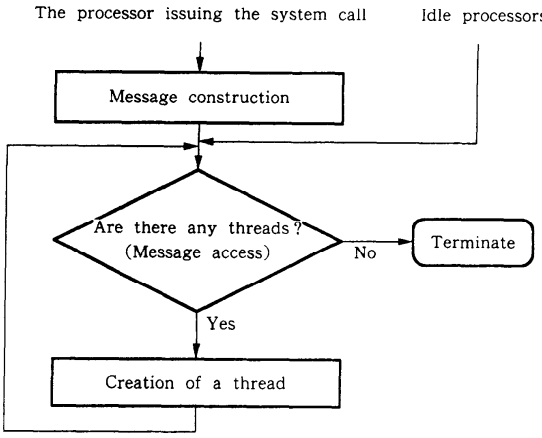


Fig. 2 Flow diagram of the simple parallel scheme.

data-structures are provided to create the threads. These structures include the free-THCBs-list, the thread ready queue, data to produce the common and the private information for each thread, and so on. Each structure is sequentially accessed. For read-only data, private cache alleviates this bottleneck. Most of the data-structures owned by the kernel, however, are read-write data. Operations on these data-structures must be mutually exclusive. Therefore, the bottleneck will be an obstacle to the performance of the kernel.

2) Memory access bottleneck: Even if multiple processors access different data-structures, these accesses are serialized when the different data-structures are stored in the same memory module.

(3) Network contention due to spin-locks

Mutual exclusion in the kernel is usually implemented with spin-locks. Processors with busy-waiting to acquire a lock, would produce excessive interconnection network traffic, and disturb the processor which holds the lock from executing the critical section.

(4) Network contention due to maintaining of cache coherence

In systems with private cache, multiple processors may perform write-access to data in the same cache block. In this case, hardware-implemented cache coherence protocols would increase network traffic.

3.4 Performance of the Simple Scheme

As described above, the performance of the simple parallel scheme for the parallel creation of multiple threads depends on tradeoffs between enhanced performance due to the parallel processing and overhead due to the message handling and memory/network contentions. We evaluate the performance of the simple parallel scheme by simulations.

3.4.1 Hardware Architecture Model

Although the operating system is for the Kyushu University Reconfigurable Parallel Processor, a hardware

architecture model to evaluate the simple parallel scheme is a common architecture to medium-size multiprocessor systems for advanced multiprocessor architectures:

1) Network architecture: Processors in systems are connected by a nonblocking interconnection network. This indicates that when two processors access different memory modules no network contention occurs. As a nonblocking interconnection network, the Kyushu University Reconfigurable Parallel Processor employs a crossbar network.

2) Memory architecture: Memory in systems consists of multiple memory modules. Each processing element has a private cache. When a cache-miss hit occurs, data of a given block size is fetched to the private cache from the corresponding memory module through the interconnection network. Cache coherence between memory and the private caches is maintained by using write-through policy. This means that current data is in the memory modules. Write-accesses to data which is not cached are performed to the corresponding memory module. A protocol to maintain cache coherence between the private caches permits only a single copy inside the caches for read-write data. This indicates that when data modified by a processor is accessed by another processor the processor must access to the corresponding memory module rather than the private cache.

3.4.2 Simulator Overview

The simulator assumes the following:

1) There are no network contentions due to spin-locks.

2) The overhead due to maintaining of the cache coherence is negligible.

3) All shared data to produce the common information and the private information for the thread is stored in a single memory module. This single-memory-module-assumption would cause memory contention.

4) Each THCB to be initialized, the message queue for the multiple-threads-creation, and the data-structures such as the free-THCBs-list and the thread ready queue, are in different memory modules: Accesses to different memory modules do not cause memory contentions.

The simulator of the simple parallel scheme consists of the following stages:

1) The message construction stage, M: The processor issuing the system call creates a message for idle processors to handle the message. The message contains the number of threads to be created. This stage is executed once when the multiple-thread-creation system call is issued.

2) The message access stage, A: An idle processor accesses the message and reads the contents. When the processor finds that there are threads to be created, the processor performs the next thread creation stage. The processing of the stage A is serialized.

3) The thread creation stage: This stage consists of the following three stages:

3-1) Stage B: The processor examines the free-THCBs-list, picks up a free THCB, and enqueues the corresponding thread ready queue. This stage may include stack allocation for the thread. The execution of this stage is mutually exclusive.

3-2) Stage C: The processor accesses the shared data of 4 bytes to produce the common information and the private information. Although each processor has the private cache, the access is performed to the single memory module rather than the private cache, because we consider the worst case that the accesses performed by multiple processors cause cache corruption. When a processor (processor A) accesses the read-write shared data cached by another processor (processor B), the cached data are invalidated. Subsequent access to the cache block performed by processor B would cause a cache-miss hit. The processing of the stage C is serialized because of the memory contention.

3-3) Stage S: The processor stores the common and the private information of 4 bytes, which is produced from the shared data in the stage C, into the THCB or the stack of the thread. The stage S can be executed in parallel.

In the simulation of the simple parallel scheme, the execution times T_M, T_A, T_B, T_C, and T_S of the stages M, A, B, C, and S are assumed to be 767, 260, 768, 78, and 69 clocks, respectively. These execution times come from assembly language of the SPARC processor for each of the stages under the following conditions:

1) An arithmetic instruction is executed in one clock.

2) Data-access-time ratio of cache memory to remote memory is 1:27.

3) When each of the stages is executed, first access to data is performed to the remote memory rather than to the cache memory. This assumption is based on the worst case where consecutive 4-bytes-data-access is performed to the remote memory because of cache corruption.

The data size of the common and the private information are assumed to be 340 bytes and 116 bytes, respectively, totally 456 bytes. Therefore, for creation of a thread, the processor executes a pair of the stages C and S 114 times after executing the stages A and B. That is, the processor executes the stages A, B, C, S, C, S, ... Figure 3 shows an example of the simulation with the simple parallel scheme, where the number of processors, N, is 3, the number of threads, th, is 7, $T_M = T_A = T_C = T_S = 1$, $T_B = 2$, and the amount of data to be initialized, D, is 8 bytes.

3.4.3 Simulation Results

Figure 4 shows the speedup rate which means the ratio of the execution time of multiple threads creation with the simple parallel scheme to that with the serial scheme where multiple threads are created by a single

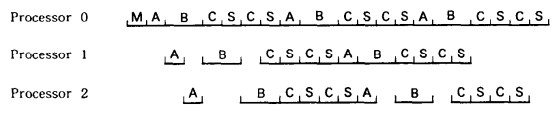


Fig. 3 An example of the simulation with the simple parallel scheme. (N=3, th=7, T_M=T_A=T_C=T_S=1, T_B=2, D=8)

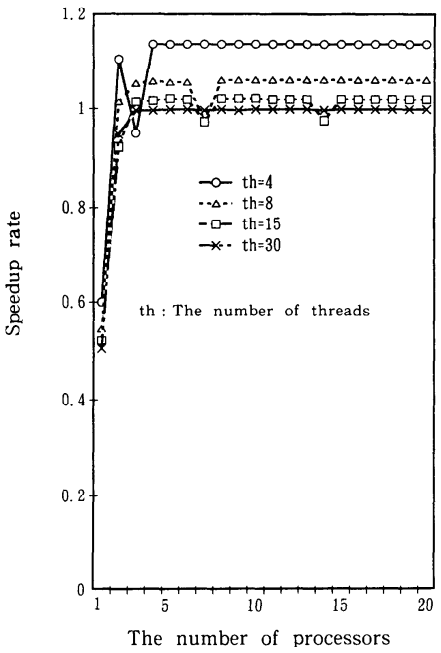


Fig. 4 Speedup rate of the simple parallel scheme.

processor. The serial scheme assumes the following:

1) The message-pool mechanism is not implemented: The message construction and the message access are not needed and there are no overheads due to the message handling.

2) The private cache is utilized: The shared data is quickly accessed.

Figure 4 says the following:

1) The speedup rate becomes saturated at the points between 1 and 1.2. The effect of the parallel creation with the simple parallel scheme is worse than expected. This comes mainly from the following:

• First, with the simple parallel scheme, the shared data is slowly accessed:

The data accesses are performed to memory because of cache corruption. On the contrary, the serial scheme allows the shared data to be quickly accessed due to the effect of the private cache.

• Second, as the number of processors increases, the execution time of a thread creation increases:

The stage C, where the shared data is accessed, is exclusively executed 114 times for each thread creation because of the single-memory-module-assumption.

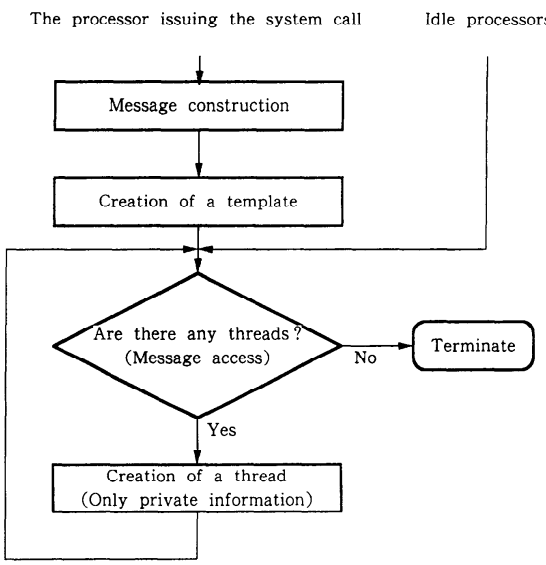


Fig. 5 Flow diagram of the parallel template scheme.

Therefore, a processor must wait for another processor to complete the execution of the stage C, and wastes CPU times. As the number of processors increases, the waiting time increases. For example, we assume that only a pair of the stages C and S is executed m times to create a thread and the execution times of these stages are the same and equal to τ. Under the assumption, when n threads are created by n processors, the execution time required for creating the thread, T, is given by $T=(m-1)n\tau+2\tau$. This equation shows that the execution time, T, increases as the number of processors, n, increases.

2) When the number of processors is small (smaller than three in Fig. 4), performance degradation occurs. This comes from the message handling overhead.

4. Improvement Schemes

As described above, the performance bottleneck mainly comes from heavy accesses to the shared data. To solve this problem, there are two approaches; i) reducing the amount of accessed data area, and/or ii) reducing the memory contention by utilizing local variables. The latter approach can utilize the private cache. According to the above two methods, we propose three schemes; a template scheme, a chunk scheme, and a combination scheme.

4.1 Template Scheme

As described in section 3.1, the information of the thread consists of two types; the common information and the private information. The template scheme allows the common information to be produced into a template once the multiple-thread-creation system call

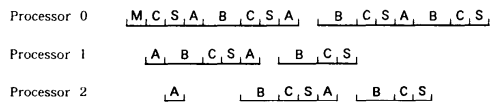


Fig. 6 An example of the simulation with the parallel template scheme. (N=3, th=7, T_M=T_A=T_C=T_S=1, T_B=2, D_C=D_P=4)

is issued, rather than whenever a thread is created (a similar scheme has been proposed in [12]). On the other hand, private information is produced whenever the thread is created. The template scheme can reduce the amount of the accessed data area. The parallel template scheme allows the template scheme to be executed in parallel; multiple threads are created in parallel, where only the private data is produced, after a single processor creates the template. Figure 5 shows the flow diagram of the parallel template scheme.

A simulation of the parallel template scheme is overviewed as the following.

The processor issuing the system call produces the common information into the template after executing the message construction stage M described in section 3.4.2. The processor executes a pair of the stages C and S described in section 3.4.2 85 times to produce the common information of 340 bytes.

Idle processors are awaken after the execution of the stage M, and create the threads. The processing of a thread creation is the same as that in the simple parallel scheme; the stages A, B, C, and S. For each thread creation, a pair of the stages C and S is executed 29 times because the amount of the private information is 116 bytes. The processors create the threads until all threads are created.

As described in section 3.4.2, the processing of the stages A, B, and C is all serialized. The stage S can be executed in parallel. Figure 6 shows an example of simulation with the parallel template scheme, where N=3, th=7, T_M=T_A=T_C=T_S=1, T_B=2, and the amount of common information, D_C, is 4 bytes, and that of private information, D_P, is 4 bytes.

4.2 Chunk Scheme

In the chunk scheme, chunk size threads are created each time of the message access. This allows the message access contention to be alleviated. In addition, with the chunk scheme, the shared data accesses can be reduced by producing local variables and utilizing the private cache. When an idle processor anticipates the multiple-threads-creation and performs the first thread creation, the processor produces the local variables for the processor and the local variables are cached. The common information is produced into some parts of the local variables. The private information is produced by using contents of some parts of the local variables, which are determined by using chunk size. For example, for the thread identifier (ID), when the processor creates the first thread of chunk size threads, the processor copies the shared thread ID into the corresponding local variable and the shared thread ID is incremented by chunk size. Caching the local variables allows the shared data access to be alleviated, and the local variables are quickly accessed.

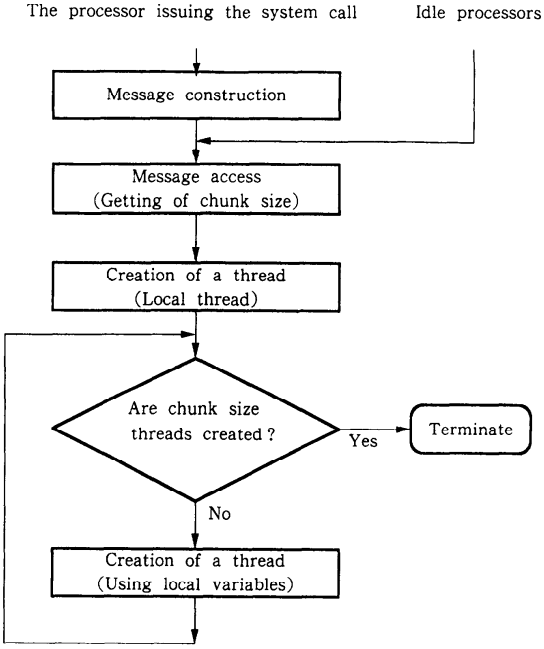


Fig. 7 Flow diagram of the chunk scheme.

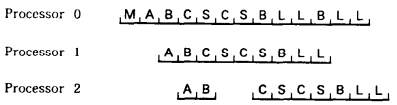


Fig. 8 An example of the simulation with the chunk scheme. (N=3, th=7, T_M=T_A=T_B=T_C=T_S=T_L=1, D=8)

In a simulation of the chunk scheme, as the local variables, the corresponding entries of the THCB which is for the first created thread (called the local thread), are used. Figure 7 shows the flow diagram of the chunk scheme. In the simulation, the chunk size is set to be $\lceil N/p \rceil$ or $\lfloor N/p \rfloor$, where N and p are the number of threads to be created and that of processors, respectively.

The simulation of the chunk scheme is performed in the following way:—

The processor issuing the system call executes the stage M. After this processing, idle processors create the chunk size threads as follows:—

1) Stage A for the message access to get the chunk size. This stage is executed once.

2) Stage B for picking up a free THCB.

3) Creation of a thread:

3-1) The first thread creation: Stages C and S are performed because of the cache corruption. For each thread creation, a pair of stages C and S are executed

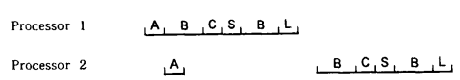


Fig. 9 An example of the simulation with the combination scheme. ($N=3$, $th=7$, $T_M=T_A=T_C=T_S=T_L=1$, $T_B=2$, $D_C=D_P=4$)

114 times.

3-2) More than the first thread creation: A new stage, L, is performed. In the stage L, the THCB is initialized by using the local variable of 4 bytes, which is in the cache memory rather than the remote memory. For each thread creation, the stage L is executed 114 times. In the simulation, the execution time of the stage L is assumed to be 70 clocks. The stage L can be executed in parallel, because the local variables which are locally-accessible are in the cache and the THCBs to be initialized are in the different memory modules from each other.

The idle processor repeats the above processing 2) and 3) after the processing 1) until all threads are created. Figure 8 shows an example of the simulation with the chunk scheme, where $N=3$, $th=7$, $T_M=T_A=T_B=T_C=T_S=T_L=1$, and $D=8$.

As one of the alternative schemes, there is a scheme where chunk size THCBs are allocated at a time. This scheme is not considered in the simulation, and is a subject for further studies.

4.3 Combination Scheme

Another improvement scheme is to combine the template scheme with the chunk scheme. In the combination scheme, when the common information is produced the template scheme is employed. For creation of the thread with only the private information, the chunk scheme is employed. This scheme has features both of the template and the chunk schemes. Figure 9 shows an example of the simulation with the combination scheme, where $N=3$, $th=7$, $T_M=T_A=T_C=T_S=T_L=1$, $T_B=2$, and $Dc=D_P=4$.

4.4 Simulation Results of Improvement Schemes

4.4.1 The Parallel Template Scheme

Figures 10(a) and (b) show the speedup rates which mean the ratio of the execution time with the parallel template scheme to that with the serial scheme described in section 3.3.3 and with a serial template scheme, respectively. In the serial template scheme, the template scheme is performed by a single processor without the message-pool mechanism. In Fig. 10(a), although the speedup curves show super-linearity until a certain value of the number of processors. This super-linearity comes from the fact of unfair comparison of the parallel template scheme with the serial scheme. These schemes have different algorithm from each other.

The speedup rate shown in Fig. 10(a) is about 3.7 for the large number of threads (e.g., 200 and 1000 in Figure 10(a)). This comes mainly from the fact that the amount of total accessed data in order to produce the common and the private information with the template scheme is reduced by four times compared to that with the serial scheme.

Figure 10(b) says that the parallel template scheme is less effective than expected compared to the serial template scheme. The speedup rate becomes saturated at points between 1 and 1.2. This mainly comes from the shared data access bottleneck. That is:

The serial template scheme utilizes the private cache. With the serial template scheme, although shared data accesses for the first thread-creation cause the cache-miss hits, the consecutive accesses are performed to the private cache. This results in fast accesses. On the other hand, with the parallel template scheme, the shared data accesses are serialized, and are performed to the memory rather than the private cache due to cache corruption.

The parallel template scheme is not as effective compared to the serial template scheme and leaves the problem of shared data access contention.

4.4.2 Chunk Scheme

Figure 11 shows the speedup rate which means the ratio of execution time with the chunk scheme to that with the serial scheme described in section 3.3.3. We can see the following:

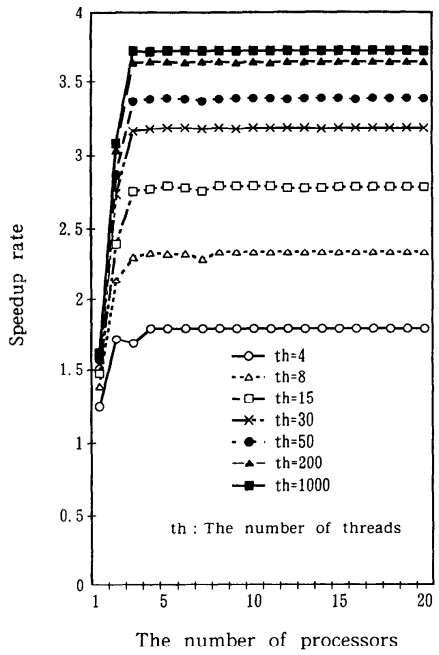
1) The speedup rate almost linearlly increases until the number of processors reaches a certain value. The certain value depends on the number of threads to be created. The maximum speedup rate is about 10.5. The speedup rate under the case of more than 2000 threads, which is not shown in Fig. 11, is almost the same as that under the case of 2000 threads. With the chunk scheme, the amount of shared-data- and message-access contentions can be reduced. In addition, the accesses in order to produce common and private information are quickly performed by the effect of private cache.

2) When the number of processors is beyond a certain value, the performance gains are degraded. The reason for this is that the shared data and the message accesses become bottlenecks under the condition where the number of processors are beyond a certain value. This means that there exists an optimum chunk size or an optimum number of processors, which depend on the number of threads to be created.

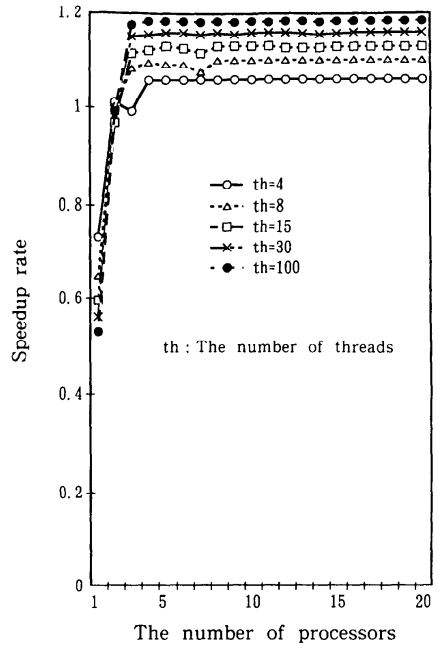
4.4.3 Combination Scheme

Figures 12(a) and (b) show the speedup rates which mean the ratio of the execution time with the combination scheme to that with the serial scheme described in section 3.3.3 and with the serial template scheme described in section 4.4.1, respectively.

As shown in Fig. 12, the combination scheme shows a tendency both to the parallel template scheme and to the chunk scheme.



(a) Comparison of the scheme with the serial scheme.



(b) Comparison of the scheme with the serial template scheme.

Fig. 10 Speedup rate of the parallel template scheme.

4.5 Considerations

(1) Issues on the parallel template scheme at schedule time

In processing of the multiple thread creation, the template scheme can reduce the amount of data to be produced by using the template. However, the template scheme would induce a new bottleneck; when the created threads are scheduled by multiple processors at the same time, the access contention to the template occurs. To alleviate the contention, we must take some approaches such as one where the template is provided by each processor. On the other hand, with the simple parallel scheme and the chunk scheme, if each THCB and stack is in different memory module the access contention at schedule time would be avoided. Thus, considering the overhead at schedule time, the parallel template scheme does not seem to be desirable.

(2) Issues on the chunk scheme

The chunk scheme is the most desirable among the proposed schemes. With the chunk scheme, there exists the optimum number of processors which participate in the multiple thread creation, and linear performance gain is attained until the optimum number. The simulation results say that the optimum numbers are between several and some tens. To get more performance gain, some allocation methods of data-structures in the kernel would be needed, as described in the next (3).

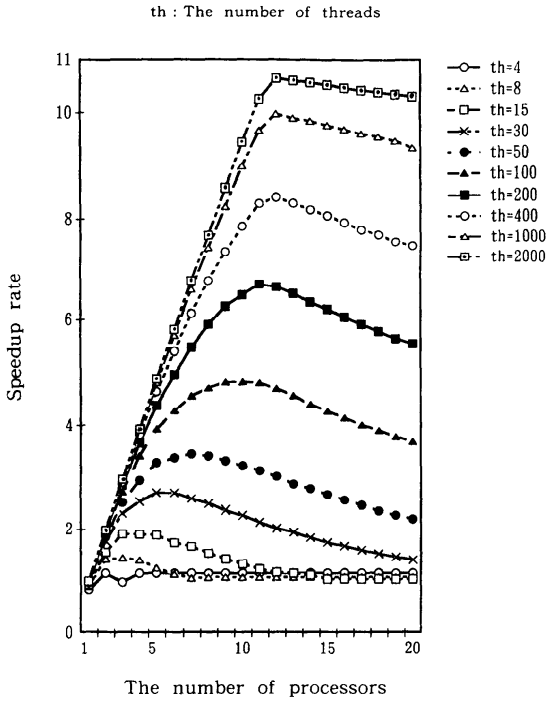
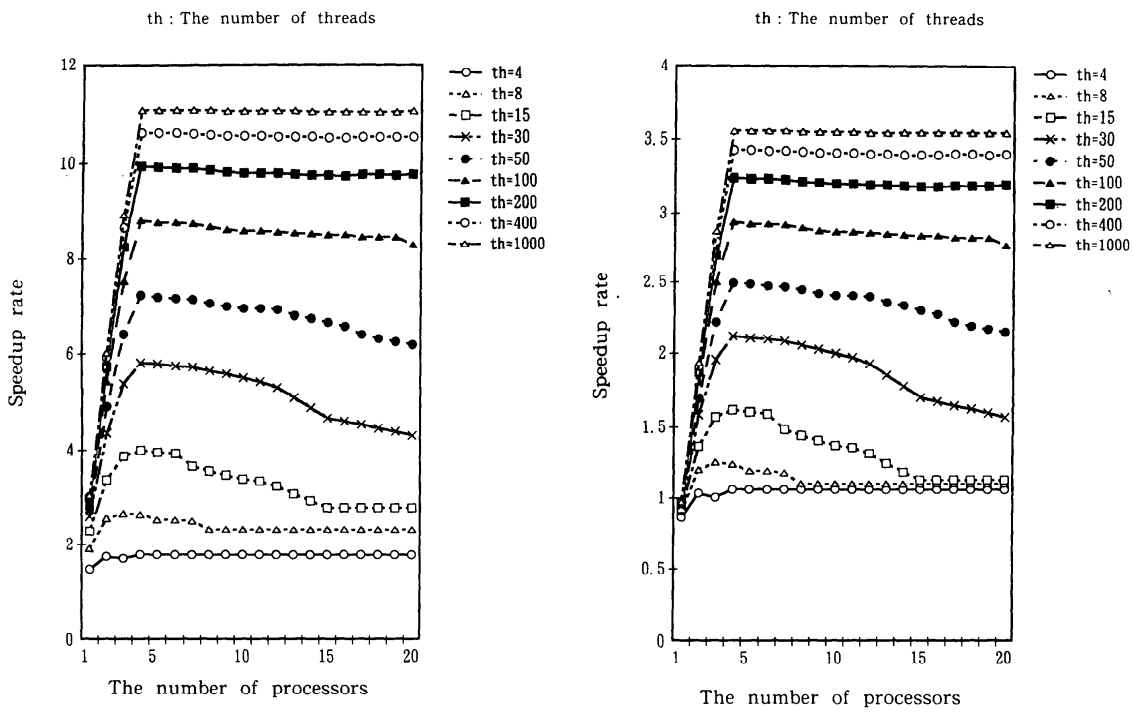


Fig. 11 Speedup rate of the chunk scheme.



(a) Comparison of the scheme with the serial scheme.

(b) Comparison of the scheme with the serial template scheme.

Fig. 12 Speedup rate of the combination scheme.

(3) Problem of the single free-THCB-list and the single thread ready queue

The proposed schemes assume that the kernel provides the single free-THCB-list and the single thread ready queue. Because the accesses to these data-structures are serialized, these single resource management methods produce another bottleneck. To alleviate the bottleneck, the data-structures must be distributed over the memory modules: A free-THCB-list and a thread ready queue are associated with a memory module. Each processor accesses different free-THCBs-list and thread ready queue. The distributed allocation method would allow the performance of the schemes to increase.

(4) Simulation results

Although the simulations assume that there is no overhead due to cache coherence protocol and no network contention due to spin-locks, we can see the essential properties of the proposed schemes by the simulations.

Overhead due to spin-locks depends on algorithms. A simple test-and-set lock shows poor performance. Queueing lock algorithms are the most scalable algorithms among those studied in [13]. As shown in Figures, parallel creation of multiple threads is applicable to the case where the number of processors amounts to some ten. For this case, the queueing lock and the ticket lock algorithms would reduce the overhead. Overhead due to hardware-based cache coherence protocol is not negligible for systems with multistage interconnection networks. However, software-based cache coherence strategies can eliminate network traffic, because caches are independently managed. To acquire accurate performance of the schemes for systems with hardware-based cache coherence strategies, the above overheads may be induced in the simulator.

We assumed the hardware model with private caches and UMA architecture. However, the characteristics of the schemes described in this paper also seem to be applicable to systems with NUMA (Non-Uniform Memory Access) architecture. For systems with bus architecture, although parallelism in the schemes would be reduced due to bus contention, the chunk scheme would also show better performance than other schemes.

5. Conclusions

The operating system under development is for the shared-memory TCMP, and aims at extracting various kinds of parallelism of the operating system itself to provide high-performance. To exploit the parallelism, the operating system is constructed by using a message-pool mechanism. A typical example of the parallelism is the parallel creation of multiple threads. As for the parallel creation schemes, we have proposed four schemes; the simple parallel scheme, the parallel template scheme, the chunk scheme, and the combination scheme.

With the simple parallel scheme, a thread is created each time the message is accessed. The parallel template scheme allows the amount of the accessed data to be reduced. With the chunk scheme, chunk size threads are created each time the message is accessed. The combination scheme combines the parallel template scheme with the chunk scheme.

We have evaluated the schemes by using simulations. Considering overhead at schedule time and from the simulation results, the chunk scheme has been shown to be the most desirable.

Acknowledgements

We would like to thank the following students who have contributed to the operating system: Y. Fukuzawa and K. Tanaka. We would also like to acknowledge the considerable contributions to the Kyushu University Rewnfigurable Parallel Processor from N. Sakuta and T. Yokota of Asia Electronics Inc., T. Yoshimoto and S. Koshiba of Bussan Electronic Systems Technology Inc., M. Takamura and K. Uchida of Fujitsu Ltd., Y. Kameda and S. Shigemura of Kyocera Corp., and S. Oyanagi and N. Tanabe of Toshiba Corp.

References

1. MURAKAMI, K., MORI, S., FUKUDA, A., SUEYOSHI, T. and TOMITA, S. The Kyushu University Reconfigurable Parallel Processor—Design of Memory and Intercommunication Architecture, *Proc. of ACM SIGARCH Int'l Conf. on Supercomputing* (June 1989), 351-360.

2. MURAKAMI, K., MORI, S., FUKUDA, A., SUEYOSHI, T. and TOMITA, S. The Kyushu University Reconfigurable Parallel Processor—Design Philosophy and Architecture—, *Proc. of IFIP 11th World Computer Congress* (August 1989), 995-1000.

3. MORI, S., MURAKAMI, K., IWATA, E., FUKUDA, A. and TOMITA, S. The Kyushu University Reconfigurable Parallel Processor—Cache Architecture and Cache Coherence Schemes—, *Proc. of Int'l Symp. on Shared Memory Multiprocessing* (April 1991), 218-229.

4. FUKUDA, A., FUKUZAWA, Y., HIROTANI, Y., MURAKAMI, K., SUEYOSHI, T. and TOMITA, S. A Parallel/Distributed Operating System for the Reconfigurable Parallel Processor (in Japanese), *JIP SIG Reports on Operating Systems, OS-43-8* (June 1989).

5. WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C. and POLLACK, F. HYDRA: The Kernel of a Multiprocessor Operating System, *Comm. ACM*, **17**, 6 (1974), 337-345.

6. JONES, A. K., CHANSLER Jr., R. J., DURHAM, I., SCHWANS, K. and VEGDAHL, S. R. StarOS, a Multiprocessor Operating System for the Support of Task Forces, *Proc. of the 7th Symp. on Operating System Principles* (1979), 117-127.

7. OUSTERHOUT, J. K., SCELZA, D. A. and SINDHU, P. S. Medusa: An Experiment in Distributed Operating System Structure, *Comm. ACM*, **23**, 2 (1980), 92-105.

8. SCOTT, M. L., LEBLANC, T. J. and MARSH, B. D. Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System. *Proc. of the 1988 Int'l Conf. on Parallel Processing* (1988), 255-262.

9. ACCETTA, M., BARON, M. R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A. and YOUNG, M. Mach: A New Kernel Foundation for UNIX Development, *Proc. USENIX 1986 Summer Conf.* (1986), 93-112.

10. TAGO, K. and MASUDA, T. A Study of Description Method for Operating Systems (in Japanese), *Trans. IPS Japan*, 25, 4 (April 1984), 524-534.

11. TAKANO, Y., TAGO, K. and MASUDA, T. Performance of a Distributed Operating System by Process Network Method (in Japanese), *Trans. IPS Japan*, **30**, 3 (March 1989), 328-338.

12. EDLER, J., GOTTLIEB, A., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., SNIR, M., TELLER, P. J. and WILSON, J. Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach, *Proc. of the 12th Annual Int'l Symp. on Computer Architecture* (1985), 126-135.

13. MELLOR-CRUMMEY, J. M. and SCOTT, M. L. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. Computer Systems*, **9**, 1 (Feb. 1991), 21-65.

(Received May 27, 1991; revised October 3, 1991)