Time Space Sharing Scheduling and Architectural Support

Atsushi Hori, Takashi Yokota, Yutaka Ishikawa, Shuichi Sakai,
Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jörg Nolte,
Hiroshi Matsuoka, Kazuaki Okamoto, Hideo Hirono

Tsukuba Research Center
Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, Japan
TEL:+81-298-53-1661, FAX:+81-298-53-1652
E-mail: {hori, yokota, ishikawa, sakai, konaka, m-maeda, tomokiyo, jon, matsuoka, okamoto, hirono}@trc.rwcp.or.jp

Abstract

In this paper, we describe a new job scheduling class, called "Time Space Sharing Scheduling" (TSSS) for dynamically partitionable parallel machines. As an instance of TSSS, we explain the "Distributed Queue Tree" (DQT) that we have proposed already. We also propose some architectural support to implement TSSS on a parallel machine as adequately as TSS on sequential machines. The most important architectural support is "network preemption." The proposed architectural support will be implemented on our RWC-1, a message-driven parallel machine, and the DQT will also be implemented in the operating system on the RWC-1, called SCore, under development in our RWC project

1 Introduction

So far, many techniques of job scheduling for partitionable parallel machines involve finding an idle partition and mapping a newly entered job onto it [3, 4, 12, 20]. In most cases, however, processor utilization is far from optimal because of fragmentation. Krueger et al. proposed a new job scheduling scheme, called "scan", and found that job scheduling order, not mapping, is more important to achieve higher processor utilization [11]. All methods, however, are batch scheduling and an interactive programming environment can not be provided. CM-5[18] and Paragon[10] provide time-sharing scheduling. In CM-5, partitioning can only be changed at system bootup time, and in Paragon (OSF/1) partitioning and the partition in which a job is executed must be specified by user.

If the target parallel machine is dynamically partitionable, one can implement a job scheduler where the parallel machine is multiplexed in time (time sharing) and divided into sub-processor-space (space sharing). With this scheduler, an interactive programming environment is possible and allocation of processor resources can be optimized because only the required processing power is allocated. The combination of time sharing and space sharing can achieve higher processor utilization than batch scheduling, because late-coming jobs may cancel the fragmentation of processor space. The scheduler should have the ability to select which partition is to be allocated to balance the load. At the same time, the scheduling process should be distributed and can run in parallel to avoid bottlenecks. The load-balancing and distributed process of scheduling are totally new. We call this kind of scheduling class "Time Space Sharing Scheduling (TSSS)".

In this paper, we focus on two aspects of TSSS. The first is TSSS (Section 2) as a new class of scheduling on dynamically partitionable parallel machines, and our Distributed Queue Tree (DQT) [9] as an instance of TSSS. The second is architectural support to realize a multi-user, multi-programming environment like UNIX (Section 3). We also describe the architectural support that will be implemented in RWC-1[16] (Section 4). One of the most important architectural support is "network preemption" derived from implementing fast process switching. We will also describe how a network

preemption mechanism is essential for parallel machines to implement not only an interactive, multi-user, and multi-programming environment, but also the detection of termination of distributed processes, global garbage collection, and checkpointing.

2 Time Space Sharing Scheduling

Time space sharing scheduling (TSSS) is a new class of job scheduling technique to provide a multi-process programming environment. It is a combination of time-sharing and space-sharing job scheduling techniques for dynamically partitionable parallel machines. Figure 1 shows an example of the TSSS. A process is a set of threads running over the processors in an assigned partition. When a process is switched, gang scheduling of the threads is assumed because the communication delay can be minimized with gang scheduling [14, 2].

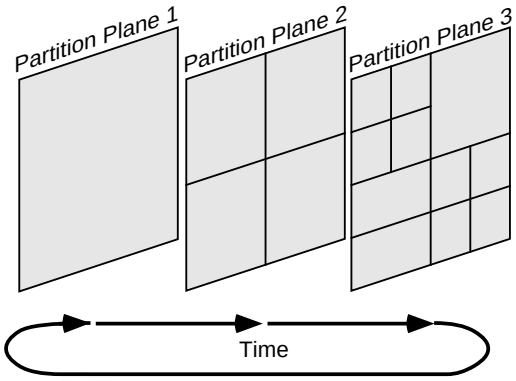


Figure 1: Example of TSSS

With TSSS, a process exclusively occupies a partition in a certain time slot. There could be a choice that each processor be multiplexed at thread level, not at process level. However, this could result in a larger working set and processor thrashing [6]. A parallel machine is multiplexed in terms of time and processor space with TSSS. A TSSS scheduler should schedule a process to each time slot and map the partition. A partition at a certain time slot is a virtualized parallel machine and a processor address space from the user's view point. We assume that the target parallel machine is homogeneous, and that the user cannot specify the partition to which a process allocated. Only the TSSS scheduler can decide which partition is allocated. The same situation can be seen in the conventional TSS, where the user cannot specify the time slot. A process group can occupy a partition. In this case, only the locations of subpartitions of the member processes can be specified, relative to the process group partition. Viewed from the operating system, a partition is a computational resource. The TSSS scheduler is also a virtual parallel machine server of various sizes. Partitions should be allocated by a TSSS scheduler according to the status of the entire system, normally, to balance the system load. The scheduling process of TSSS, however, should be distributed to avoid possible bottlenecks.

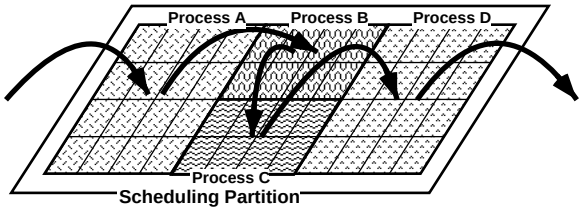


Figure 2: Example of Piped Processes

A TSSS system provides not only enough computational power for the job (task), but also an interactive programming environment by time-sharing. Processes can also run in parallel, if they are located in different but non-overlapped partitions. A good example of the power of TSSS can be found in piped processes in UNIX. In Figure 2, process A, B, C, and D are connected with pipes (denoted as curved arrows) and located in disjoint subpartitions in a scheduling partition. These four processes can be scheduled simultaneously and run in true parallel. The input of process A can be a terminal, and this piped command can be interactive. In UNIX, the speed of this piped command is tentative and is dominated by the slowest process in the command. With TSSS, however, the speed of all processes can be equalized and can reach maximum speed when the partition sizes of the processes are chosen in the right way. Thus the benefit of command modularity and the best parallel performance execution can be gained in a familiar and natural manner.

2.1 Distributed Queue Tree

We proposed the Distributed Queue Tree (DQT) [9] as an instance of TSSS. The DQT is a distributed tree structure for process scheduling management. Each node of the DQT has a process run queue. Every process in the queue requires that the number of processors does not exceed the partition size of the node. The DQT tree

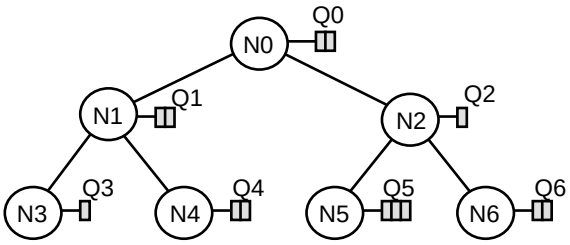


Figure 3: Example of DQT

Table 1: Example of DQT scheduling

Time Slot #	PE0	PE1	PE2	PE3
0	$Q_0(0)$			
1	$Q_0(1)$			
2	$Q_1(0)$		$Q_2(0)$	
3	$Q_1(1)$		$Q_5(0)$	$Q_6(0)$
4	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(1)$
5	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(0)$
6	$Q_0(0)$			
7	$Q_0(1)$			
8	$Q_1(0)$		$Q_2(0)$	
9	$Q_1(1)$		$Q_5(0)$	$Q_6(1)$
10	$Q_3(0)$	$Q_4(0)$	$Q_5(1)$	$Q_6(0)$
11	$Q_3(0)$	$Q_4(1)$	$Q_5(2)$	$Q_6(1)$
12	$Q_0(0)$			
:	:			

structure should reflect the nesting of the dynamic partitioning. Each DQT node should be distributed to the processor in the partition corresponding to the node. Each DQT node only communicates with its supernode and subnodes. When a process is suspended, the process should be dequeued from the process run queue. In the DQT, this queue operation is needed only in a processor that plays the role of a DQT node.

Figure 3 shows an example DQT. Each node has a process run queue represented by a rectangle to the right of the node. The width of each rectangle is equal to the length of the queue. The root node, N0, is responsible for the entire processor space (full partition). Each of nodes N1 and N2 is responsible for a halved partition. Each of nodes N3, N4, N5 and N6 is responsible for a quartered partition.

Table 1 shows the scheduling corresponding to the DQT in Figure 3. In this table, the jth process in the queue Q_i of the ith node is denoted by "$Q_i(j)$". The entire processor space is assigned to $Q_0(0)$ at time slot 0 and to $Q_0(1)$ at time slot 1. In time slot 2, halved

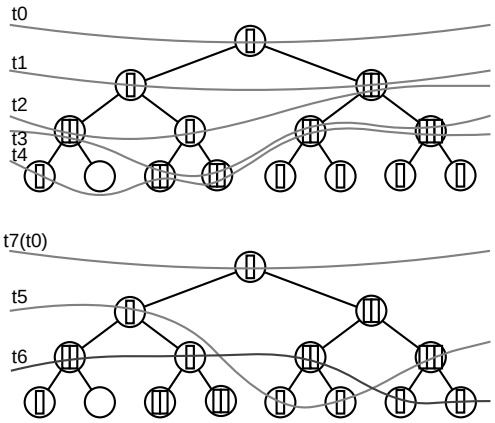


Figure 4: Example of front movement

partitions are assigned and two processes are running simultaneously in the adjacent partitions. In time slot 3, the right-hand side halved partition is halved again, while the left-hand side halved partition is left as is since there are two processes in queue Q_1. Every process is scheduled at least once in 6 time slots in this case.

A DQT node is activated when a process in the queue of the node is scheduled. At a certain time, the line connecting activated DQT nodes in a tree diagram is called a **front**. Figure 4 shows an example of movement of the front. In this figure, the rectangles in the DQT nodes represent the process run queue in that node. If the load of DQT is well-balanced, the front is a horizontal line moving downward repeatedly. The lines $t0$ and $t1$ in Figure 4 are examples. The front moves faster on the DQT branch with the lighter load than on the heavier loaded branch (The front is denoted by $t2$ or $t3$, for example). If a part of the front hits the bottom of the tree ($t4$), then the part goes back to the node where the load is unbalanced ($t5$ and $t6$). This is to keep the processors as busy as possible. Consequently smaller processes may be scheduled more often than larger processes. As supposed, this strategy can cause an unfairness in scheduling.

The policy of deciding which partition is to be allocated when a process is created is very important in balancing the DQT load. This is because job allocation is the only chance to balance the load of a DQT. A well-balanced DQT exhibits not only good processor utilization, but also a shorter response time and fair scheduling [9]. We proposed various job allocation policies [9]. Figure 5 shows an example of one of the proposed job allocation policies.

The number on each DQT node represents the load

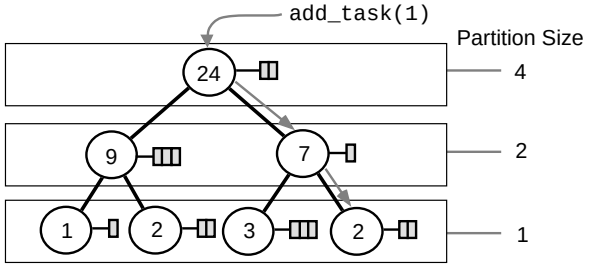


Figure 5: Example of job allocation

on the branch, in this case, the number of virtual processors needed to schedule all processes in the sub-DQT at once. For example, the number on the left node in the second level is 9, because the node itself requires 6 virtual processors (three processes times partition size two) and the virtual processors in the subnodes (one for left subnode and two for right subnode). The `add_task` message is sent to the root when a partition for a newly created process is required. This message is forwarded to the subnode whose load is lighter, until it reaches the node that has the required partition size (in this case, the required size is one). The goal of the `add_task` message will be the partition for a newly created process.

We have been evaluating DQT with a number of simulations. The latest simulation results show that DQT exhibits very good linearity in processor utilization from low-load situations to high-load situations, independent of job size distribution, and also good stability even with a 99% workload.

2.2 Related Works

Feitelson and Rudolph proposed a "Distributed Hierarchical Control" (DHC)[7, 6] that is also an instance of TSSS. DQT and DHC are very similar in the nature of distribution and scalability. The major difference between DQT and DHC, however, lies in the assumptions made and the target computing environment. In DQT, the scheduling unit is a "process", an entity of parallel program execution and a set of thread, while in DHC the unit is a "thread" as a part of a parallel program execution on a single processor. In DHC each scheduler at the lowest level should manage the threads on each processor. In DQT, however, each DQT node, including the DQT nodes at the lowest level, only manages a "virtualized parallel machine" corresponding to the node. The DQT nodes at the lowest level can be associated with the smallest partition size larger than one. Thus the height (number of levels) of DQT is the magnitude of the partition size that the system provides. The thread management in DQT is orthogonal to job scheduling.

The disadvantage in the assumption of DQT is that the number of running processors (threads) may vary during program execution. If the number of running processors is less than partition size, then processor utilization is decreased. The overhead of forking threads over processors, however, can be much lower than the case in DHC because there is no scheduling overhead. While in DHC, idle processors can be eliminated because of thread level scheduling, the scheduling overhead can be considerably heavy when the lifetime of threads is relatively short. Thus DHC is somewhat aimed at a distributed computing environment in which a longer thread lifetime assumed, but our DQT is targeted at more closely connected parallel machines in which a shorter thread lifetime supposed.

We assume that the overhead of thread invocation is very low, and that there is some architectural support for partitioning and gang-scheduling. In message-driven, multi-threaded processors like MDP [5] or our RWC-1 [16], threads are controlled by hardware. The software overhead in thread control should be eliminated.

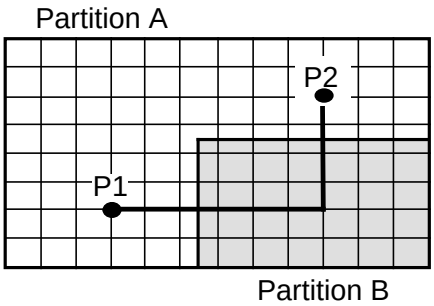
3 Architectural Support for TSSS

Some kind of architectural support is required to implement efficient TSSS. In the following subsections, we discuss the kind of architectural support that enables practical TSSS. In subsections 3.1 and 3.2, we concentrate on the partitioning itself. We discuss the characteristics of partitioning that enables efficient TSSS. In subsection 3.3, we propose a new concept called "network preemption."

3.1 Degree of partitioning

A number of network topologies have been proposed so far. Most of the them, however, are mainly considered in terms of the application or architecture, but not in terms of the parallel operating system. A parallel application can be aware of network topology. Job scheduling, however, is a part of the operating system and it is relatively hard to maintain. The control structure of TSSS should not be aware of network topology from the viewpoint of portability.

In practice, the TSSS control structure is not completely independent from the network topology. Here we consider the "partitioning degree" of a network. For



The message of partition A being passed from node P1 to node P2 passes through partition B (in case of X-Y routing). Partition A is open and partition B is closed.

Figure 6: An open partition and a closed partition in a 2D mesh

example, the partitioning degree of a hyper-cube is usually two, because every sub-partition is halved recursively. In a 2D-mesh, the partitioning degree can be four. The partition allocation problem is very similar to the problem of memory allocation. If the distribution of the required partition size is unknown, a partitioning degree of two and a binary buddy allocation strategy are the best from the viewpoint of fragmentation and the simplicity of algorithm [15]. This gives a new approach to the design of a network topology on a parallel machine. In a 2D-mesh, the partitioning degree can be two. In this case, however, both the mean and maximum hop counts increase, and hot spots can appear more easily. Thus, parallel machine designers should consider the partitioning degree if they want to provide TSSS.

3.2 Open and closed partition

The partitions of a network can be divided into "open partitions" and "closed partitions", depending on the network topology, routing algorithm, and shape of the partition.

Open partition: On a direct network where every router is associated with a processor, if any kind of message can go out the partition that includes the sending node, then the partition is called an open partition.

Closed partition: In communications between processors in the same partition, if no messages goes out of the partition, then the partition is called a closed partition.

In an open partition, a process may interfere or block the execution of another process when it sends a number of messages into the network. This situation should be avoided when a time-critical application, such as a multimedia server, runs simultaneously with other applications. Thus, all possible partitions should be closed to prevent any inter-process interference. In terms of fault tolerance, a defective processor or router can be easily isolated, if all possible partitions are closed.

3.3 Network preemption

The process switching speed is the key, implementing TSSS as efficiently as on sequential machines. If time-critical applications are the target of the machine, then the process switching time should be fast enough, and should guarantee the maximum process switching time.

On a distributed memory parallel machine, the handling of messages being passed around a network is the major issue faced in guaranteeing the process switching time. In CM-5, time sharing in a partition is implemented with an AFD (All Fall Down) operation [18]. When a scheduler decides to switch a process, the subnetwork in the partition enters AFD mode. In this mode, all messages in the subnetwork go to the nearest processors regardless of the message destinations. After the AFD mode, the kernel switches to a new process. The new process's messages that were previously saved by the AFD, are sent into the network again. The message order is not preserved. Further, in CM-5, message sending in user-mode can fail when a kernel sends a message, or when process switching takes place [19]. Therefore, any message sending in user-mode should always be verified.

We will generalize this AFD scheme. A typical question would be "Why not preempt the network as well as the processors ?", since the network is a crucial part in a parallel machine. There is a big difference between AFD and network preemption. With network preemption, the message sending in user-mode never fails, and the message order can also be preserved.

On a direct network, network preemption can be implemented more easily than on an indirect network if all possible partitioning of the network is closed. This is because every router is associated with a processor and they are close enough to implement the network preemption mechanism in a direct network. In a closed partition, all router statuses must be saved when switching processes are in the partition. Thus, closed partitioning guarantees the locality of process switching. The time to preempt a (sub)network may not depend on system size (number processor or router) and may be a constant order.

With closed partitions, one can avoid inter-process interference caused by hardware conflicts. With a network preemption mechanism, even in a situation in which a user sends a number of messages into a network and the network is saturating, the maximum process switching time can be guaranteed to be reasonable. Network preemption also means avoiding inter-process interference in the time domain.

A network preemption mechanism provides not only fast process switching, but also termination (idle) detection of a distributed computation. A distributed computation is said to be terminated, if (i) every process is idle, and (ii) there are no messages in a network [1]. The second condition can be checked by investigating on the saved router status. In practice the next condition is needed: (iii) no suspending systemcalls. This checks the existence of one or more threads waiting for the result(s) of systemcall(s), such as I/O completion. If only condition (iii) fails, then the process is idle. The simplest way to check if a process is idle or not is to inspect the router status sampled at the every process switching. The same technique can be used in global garbage collection (GC), since the distributed termination problem and global GC problem are dual [17].

The other application of network preemption is checkpointing. With network preemption, the process context and router status can be stored onto disk(s) at a checkpoint, and the status can be restored. Then, program execution can be restarted. Fast process switching with a network preemption mechanism helps to implement efficient and practical checkpointing. Thus a network preemption mechanism is considered to be essential for a practical parallel machine.

The synchronization of network preemption now becomes an issue. If a network preemption signal is propagated by a normal multicasted message, some of it can be lost. This is because some of the multicasted messages may be saved with their router statuses into the processors' memory, before the signal delivery to every router in a closed partition has been completed. This situation may cause a deadlock or failure of synchronization of network preemption. To prevent this, special broadcast and synchronization mechanisms are needed to implement network preemption.

3.4 Related Works

Lin and Wu proposed a conflict-free network [13] where any hardware resource contentions can never arise in communications. Here, the hardware resources may include wires, message buffers, input ports, output ports, and so on. If any communications in a partition are conflict-free from the communications in other partitions, then the interference between processes in different partitions can be avoided. In a direct network, the closed partitions are conflict-free of each other. However, as described in subsection 3.2, the conflict-free partitions may not be enough for implementing a practical multi-user, multi-program environment.

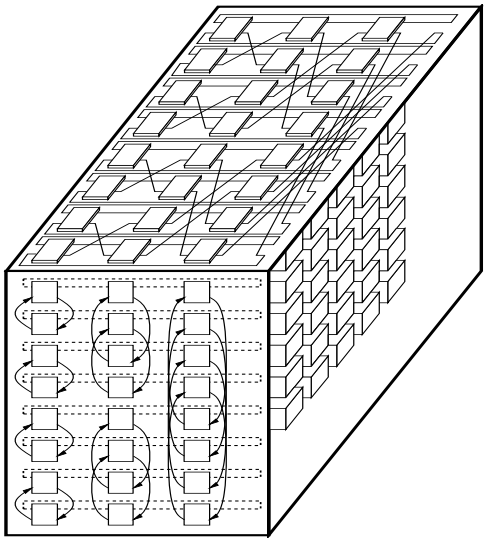


Figure 7: Example of CCCB

4 Architectural Support in RWC-1

RWC-1[16] is a message-driven, multithreaded parallel machine under development in our RWC project. An operating system kernel for RWC-1, named "SCore,"[8] is also under development. The DQT scheduling will be implemented on the SCore. RWC-1 will have architectural support for TSSS, as described in the previous section.

The network is called "Cube Connected Circular Banyan" (CCCB). Figure 7 shows an example of a CCCB network. Every RWC-1 router has a virtual-cut-through routing mechanism which is deadlock free. The FIFO message order is preserved. The partitioning nature of CCCB is almost the same as in a hyper-cube. CCCB has the characteristics of a partitioning degree of two, with all possible partitioning being closed.

The network preemption mechanism is called a **drain** as opposed to the AFD of CM-5. When a drain signal is generated from a processor, the signal propagates to every processor in the current partition. To enable this, at

least every router on the "edge" of the partition should know which ports to propagate the drain signal to. The drain signal is completely different from a normal communication message, but is more like a control signal. Upon receiving the drain signal, every router freezes the receiving and sending of messages as soon as possible, and propagates drain signals to the next routers. When every input port of a router falls into the drain mode, the router asserts an interrupt to let the local kernel know that now is the time to switch processes. Thus drain signals are synchronized and gang-scheduling of threads can be realized with the interrupts triggered by the synchronization.

The procedure to switch processes can be outlined as follows:

1. The scheduler decides to switch processes.
2. **The scheduler transmits a drain signal to the current partition. Every router in the partition is now frozen.**
3. **Every local kernel receives an interrupt triggered by the drain signal.**
4. **The kernel saves the router status.**
5. The kernel saves the process status.
6. The kernel restores the process status.
7. **The kernel restores the router status.**
8. **The kernel sets partition information to the router (if partitioning is changed).**
9. **The kernel releases the router. The router is now operational.**
10. The kernel restarts the new process.

We will now try to measure the cost of the process switch on RWC-1. In the procedure above, those steps in **bold** are the additional costs to the sequential machine. The cost of gang-scheduling with this drain mechanism is dominated by two procedures; one is the cost of propagating the drain signal, and the other is the cost of saving and restoring the router status.

With the drain mechanism of RWC-1, the theoretical maximum time to propagate the drain signal (T_{drain}) can be estimated by

$$T_{drain} \leq D_{max} \times L_{max} + D_{max} \times H_{max}$$

where D_{max} is the maximum hop count in the largest partition, L_{max} is the maximum length of the message, and H_{max} is the maximum time for one hop. The first term on the right hand side is the maximum time to end the receiving of the incoming message. Since in RWC-1, messages are transmitted and received in a pipeline fashion, one must wait for the receiving of the incoming message (or packet) to complete. The second term is the time to propagate the drain signal.

The time to save or restore the router status depends mostly on the size of the buffer in the router. The total size of the buffer in an RWC-1 router is relatively large, because of virtual-cut-through routing and the avoidance of deadlock. To shorten the process switching time, we designed RWC-1 so that saving and restoring the router status can be done by hardware in the background. Thus, the operating system can devote itself to saving or restoring the process status.

In the RWC-1 with 1,024 processors, the T_{drain} is estimated to be less than 200 clock cycles and the saving or restoration time of the router status is estimated to be less than 2,000 clock cycles. Including the operating system overhead, the process switching time is expected to be less than 500 μs. This is far less than the 4 ms processes switching time on CM-5 [2].

5 Summary

We described a new job scheduling class called "Time Space Sharing Scheduling" for dynamically partitionable parallel machines. TSSS can implement an interactive, multi-user, multi-programming environment for parallel machines that is as adequate as conventional sequential machines. As an instance of TSSS, we have explained our "Distributed Queue Tree." We also suggested that parallel machine designers be aware of architectural support for (i) partitioning degree being two, (ii) closed partitioning, and (iii) network preemption in implementing a practical TSSS. We also described the architectural support on the RWC-1. The process switching ability of RWC-1 will be comparable with sequential machines.

Among the proposed architectural support, network preemption is the most important concept for parallel machines. Because it not only provides fast process switching, but can also be used to detect the idle or terminated status of a distributed process, checkpointing, and global GC. We believe that most parallel machines in the future should implement the mechanism of network preemption.

With the architectural support for TSSS and DQT scheduling, we believe that TSSS will become one of the most practical job schedulings schemes for parallel machines. We will implement the proposed architectural support in RWC-1, and DQT in the operating system

for RWC-1, SCore.

References

[1] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *Computing Surveys*, 23(1):49–90, March 1991.

[2] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Supercomputing'94*, pages 590–599, November 1994.

[3] M.-S. Chen and K. G. Shin. Subcube Allocation and Task Migration in Hypercube Multiprocessors. *IEEE Transactions on Computers*, 39(9):1146–1155, 1990.

[4] P.-J. Chuang and N.-F. Tzeng. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. *IEEE Transactions on Computers*, 41(4):467–479, 1992.

[5] W. J. Dally, J. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, and P. R. Nuth. The Message-Driven Procesor: A Multicomputer Processing node with Efficient Mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[6] D. G. Feitelson and L. Rudolph. Distributed Hierarchical Control for Parallel Processing. *COMPUTER*, pages 65–77, May 1990.

[7] D. G. Feitelson and L. Rudolph. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control. In *International Conference on Parallel Processing*, volume I, pages 1–8, 1990.

[8] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo. Overview of Massively Parallel Operating System Kernel SCore. Technical Report TR-93003, Real World Computing Partnership, 1993.

[9] A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo. A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Science*, volume II, pages 173–182. IEEE Computer Society Press, January 1995.

[10] Intel Corporation. *PARAGON OSF/1 USER'S GUIDE*, April 1993.

[11] P. Krueger, T.-H. Lai, and V. A. Dixit-Radiya. Job Scheduling Is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):488–497, 1994.

[12] K. Li and K.-H. Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, 12(5):79–83, May 1991.

[13] W. Lin and C.-L. Wu. A Distributed Resource Management Mechanism for a Partitionable Multiprocessor System. *IEEE Transactions on Computers*, 37(2):201–210, February 1988.

[14] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[15] J. L. Peterson and T. A. Norman. Buddy System. *Communication of the ACM*, 20(6):421–431, June 1977.

[16] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, and M. Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Proceedings of 1993 International Conference on Supercomputing*, pages 251–260, 1993.

[17] G. Tel and F. Mattern. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[18] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, November 1992.

[19] Thinking Machines Corporation. *NI Systems Programming*, October 1992. Version 7.1.

[20] Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.