Implementing the Combination of Time Sharing and Space Sharing on AP/Linux

Kuniyasu Suzaki[1,2] and David Walsh[1]

[1] Australian National University, Canberra, ACT 0200, Australia
{suzaki, dwalsh}@cafe.anu.edu.au
http://cap.anu.edu.au/cap/projects/linux/
[2] Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba, 305, Japan

Abstract. We report the implementation of a scheduling method which combines time sharing and space sharing on AP/Linux. To run many tasks simultaneously on a parallel computer, the parallel computer system needs a partitioning algorithm that can partition processors for incoming tasks. However, a typical problem for the algorithm is a blockade situation, which causes low processor utilization and slow response. To avoid such a situation, we present a Time Sharing System(TSS) scheme that uses a partitioning algorithm. In this paper we state the implementation design of our TSS on a real parallel computer, the Fujitsu AP1000+. The design is based on the parallel operating system, AP/Linux. We report our current implementation and the performance.

Keywords: time sharing, space sharing, partitioning algorithm, AP/Linux

1 Introduction

Parallel computers are becoming more popular for many applications and many commercial parallel computers are on the market. Since every application cannot utilize all processors in a parallel computer, it is desirable to run many tasks simultaneously. To achieve this, we can use partitioning algorithms, which allocate a region of processors for an incoming task and then release the region when the task is outgoing, and do not allow the regions to overlap.

Partitioning algorithms have been proposed for many architectures. For mesh-connected parallel computers, these algorithms include the Frame Slide[1], the Two-Dimensional Buddy[2], the First Fit[3], the Best Fit[3], the Adaptive Scan[4], the Busy List[5], the Quick Allocation[6], and the non-partitioning algorithm[7, 8]. However, such partitioning algorithms have a typical drawback, namely, the blockade situation.

Figure 1 illustrates the blockade situation occurring on a mesh-connected parallel computer. Since such partitioning algorithms have a first-come-first-served(FCFS) policy, their incoming tasks(1 and 2 in Figure 1) are allocated in the incoming order and can be accommodated at the same time. When task 3 is incoming, it cannot be allocated, because the allocation is prevented by tasks

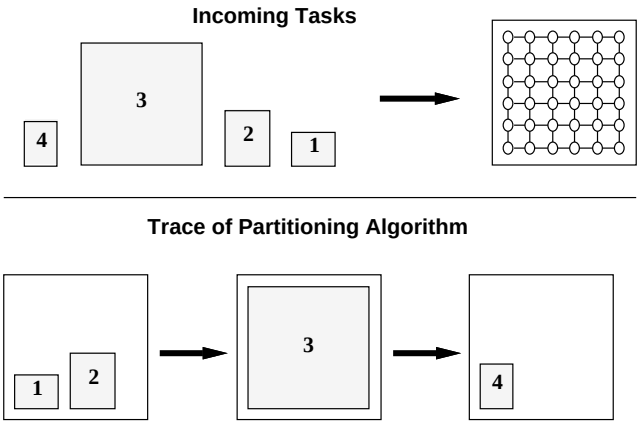

Fig. 1. Blockade situation for a mesh-connected parallel computer.

1 and 2. Task 3 is then queued until both tasks 1 and 2 are outgoing. Even if task 3 is allocated, task 4 cannot be allocated; thus, causing the blockade situation. If task 3 did not exist, tasks 1, 2, and 4 would be allocated on the parallel computer at the same time. The blockade situation causes decrease in the processor utilization and delay in response time for each task.

To avoid this blockade situation, Time Sharing System(TSS) schemes that uses a partitioning algorithm are proposed by Yoo et al.[9] and Suzaki et al.[10] as a kind of gang scheduling[11],[12]. They describe more detail of relation of space sharing and time sharing on a mesh-connected parallel computer.

The TSS provides virtual parallel computers which are activated alternately on a real parallel computer.Tasks are allocated on virtual parallel computers by a partitioning algorithm. In Figure 1, tasks 1, 2, and 4 are allocated on one virtual parallel computer and task 3 is allocated on another. TSS can avoid the blockade situation. It also achieves high processor utilization and quick response for each task.

In this paper we state the implementation design of TSS on a parallel computer, the Fujitsu AP1000+. The design is based on the parallel operating system AP/Linux[13] which is developed by the CAP group at the Australian National University(ANU). According to the design we have implemented the TSS on the AP1000+ at the ANU and at Electrotechincal Laboratory(ETL), which have 16 CPU's. We also confirmed performance of the tasks under our TSS.

In the next section, we present a number of time sharing systems which use partitioning algorithms. In Section 3, we give an overview of the Fujitsu AP1000+. In Section 4, we introduce AP/Linux, a parallel operating system for the AP1000+. In Section 5, we show the design of TSS on the AP/Linux. We report the current status of our implementation and performance in Section 6.

In Section 7, we discuss our future work. Finally, in Section 8, we state our conclusions.

2 Time Sharing System on Parallel Computers

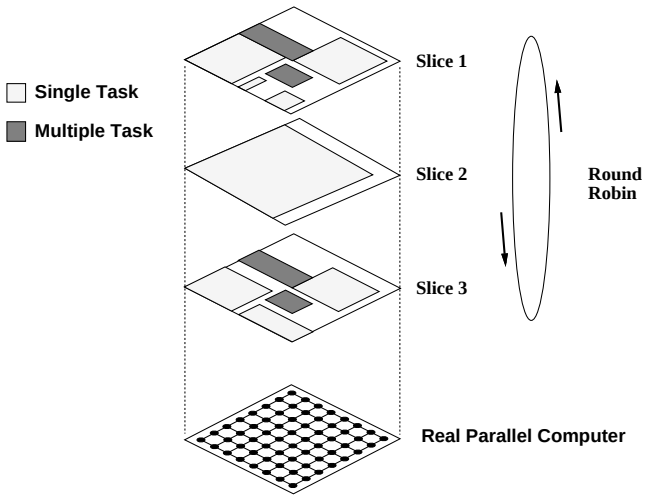

Fig. 2. TSS diagram showing single and multiple tasks.

On a single computer, a TSS allocates CPU time to tasks alternately, thus improving the response time of a task. This technique is also available for parallel computers. For example the CM5[14] offers TSS for users. However, there are idle processors, because not every application can utilize all the processors in a parallel computer. To decrease the number of idle processors, the use of partitioning algorithms have been proposed in the papers[9],[10]. The TSS scheme is combination of time sharing and space sharing, and is able to avoid the blockade situation, which is a typical problem in partitioning algorithms.

The TSS provides a number of virtual parallel computers that can be mapped to the structure of the target parallel computer. We call such virtual parallel computers *slices*. An incoming task is allocated on a slice. If no room is left for the incoming task on existing slices, a new slice is created and the task is allocated on this new slice. Based on round-robin scheme, each slice is alternately activated on the real parallel computer.

The TSS allows tasks to be allocated on more than one slice. If a processor region for a task on a slice is free on other slices, the task can sit on these other slices. The task is then executed as many times as the number of slices that hold

that task, while all slices are activated by a round-robin scheme. We call a task that exists on more than one slices a *multiple task*, and a task that exists on only one slice a *single task*. Figure 2 illustrates multiple tasks and single tasks on slices. The idea of multiple task is resemble to multiple slot[15] and dynamic partitioning[16]. The multiple task is specialized for a mesh-connected parallel computer and is considered to searching algorithm of multiple tasks.

The tasks on slices are managed using the data structure illustrated in Figure 3. Each slice has two lists that manage the tasks, a *single task list* and a *multiple task list*. A single task list has submesh information for single tasks; namely, the x and y starting points of the submesh, the submesh width, and the submesh height. A multiple task list has the same submesh information, and also includes information for multiple tasks; namely, the slice number which links multiple tasks. The information for multiple tasks is used for the termination of tasks and reduction of slices[10].

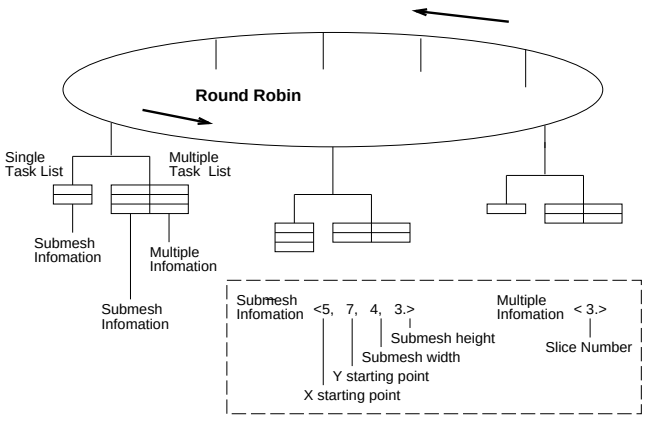

Fig. 3. Data structure for slices.

3 AP1000+

The AP1000+ is a distributed memory parallel computer, developed by Fujitsu. It has three networks; a broadcast network(B-net; 25MB/s), a torus network(T-net; 50MB/s per a link), and a synchronization network(S-net) illustrated in Figure 4. On the AP1000+ a processing unit is called a *cell*. Each cell is controlled by a host machine using the B-net and S-net. The AP1000+ offers special instructions which allow remote memory access. The remote memory access does not disturb a remote processor. Each cell can read remote memory with a *get* instruction and write with a *put* instruction.

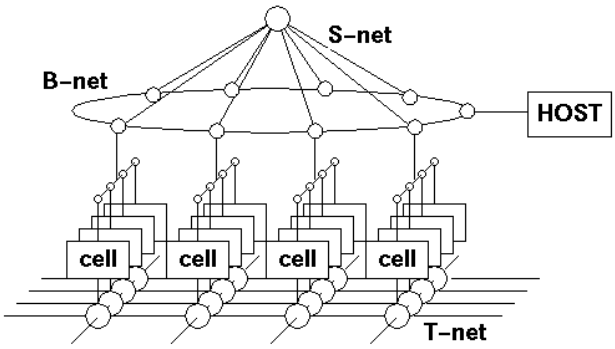

Fig. 4. AP1000+

Figure 5 shows the detail of a cell. The cell consists of a SuperSPARC processor, a memory controller(MC), a message controller(MSC+), a routing controller(RTC), and a B-net interface(BIF). The SuperSPARC offers write through cache action for memory protection due to the remote memory access. The MC has a MMU and manages address translation. The MSC+ controls messaging and remote memory accesses. The messages on the T-net are forwarded by worm hole routing. The BIF manages broadcast of messages using the B-net and synchronization using the S-net.

Fujitsu offers a simple operating system called "Cell-OS" for the AP1000+. Under Cell-OS, the machine is reset before launching each parallel program, and the kernel is loaded with the parallel program. Cell-OS offers a single-user and single-program environment and does not support all UNIX compatible functions. To improve this environment, AP/Linux has been developed by the CAP group at the Australian National University.

4 AP/Linux

AP/Linux[13] is the operating system for the Fujitsu AP1000+. It is based on the popular *Linux* operating system. It provides the facility of a parallel programming environment, as well as normal unix functionality. The Linux kernel is loaded on each cell and is long lived. It manages process scheduling, virtual memory, file systems, and system calls. Normal user logins are permitted on any cell and it behaves as a modern operating system.

AP/Linux also offers the environment to run parallel programs. We can run parallel programs on AP/Linux. To build parallel applications, AP/Linux offers APlib and MPI libraries[17]. APlib library offers a compatible Cell-OS interface. APlib and MPI libraries are able to access the MSC+ hardware directly from user space and give high throughput and low latency.

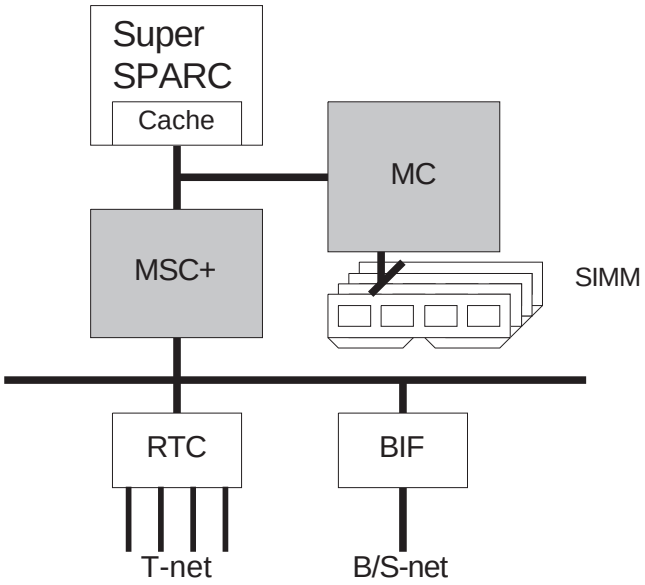

Fig. 5. Cell

The parallel programming environment is supported by the parallel run command *prun*, the parallel daemon *paralleld* which exists on each cell and manages the creation and termination of parallel processes, and special scheduling for parallel processing in the kernel(Figure 6). The normal scheduling provided by the Linux kernel can also manage parallel processes. However, performance can be poor due to the time waiting for messages, particularly when the message libraries wait by polling. The authors of AP/Linux recommend that a cooperative scheduling is used to achieve reasonable performance[13].

The procedure to run parallel programs is divided into process allocation and synchronization. We state these details.

4.1 Process allocation

prun is the command used to launch a parallel process. It requires the number of processors and a parallel program name. For example, in Figure 6 the three *prun* commands require 2×2 processors to run *task1*, 3×3 processors for *task2*, and 1×3 for *task3*. A *prun* command can be issued on any cell and it sends a message to the *paralleld* on the primary cell. In this paper we assume that the primary cell is cell0.

The *parallelds* manage creation and termination of parallel processes and manage a standard I/O session between *prun* and the parallel process. The *paralleld* on cell0 reserves processors required by the *prun* and sends the par-

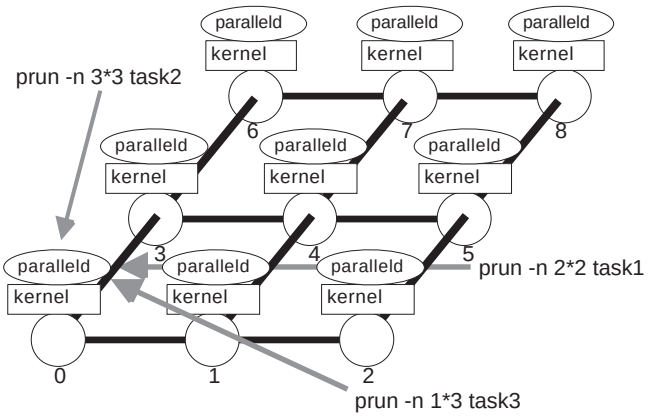

Fig. 6. AP/Linux

allel process image to the *parallelds* on the reserved processors. Each *paralleld* launches the parallel process using the *clone()* system call and the parallel process enqueued to each local scheduling queue. The *clone()* system call can get the same process ID(PID) and Task ID(TID) on each cell. The IDs make it possible to identify the parallel process on all cells. On AP/Linux, high numbered PIDs are reserved for parallel processes and low numbered PIDs are reserved for single processes. The TID is attached to the header of messages and is used for posting the message to the destination process.

Figure 7 shows the process allocation required by *prun*s in Figure 6. The processors which are offered by the *paralleld* start from cell0. The scheduling queue on cell0 has all parallel processes. In the figure, cell0 has three parallel processes although cell5,6,7, and 8 have only one parallel process. This situation is caused by lack of space sharing, which results in poor load balancing over the machine.

When a parallel process is terminated, the *paralleld* catches a signal from the parallel process and cuts the standard I/O session between the parallel process and the *prun*.

4.2 Synchronization

AP/Linux provides a loose wake-up synchronization of parallel processes. It is a reasonable compromise, because it is difficult to predict the state of processes on remote processors in UNIX environments, including Linux. For example, page faults are done asynchronously.

The local scheduler on cell0 manages wake-up synchronization for each parallel process, because all parallel processes are allocated from cell0, and the local scheduling queue has all parallel process information. The local scheduler on

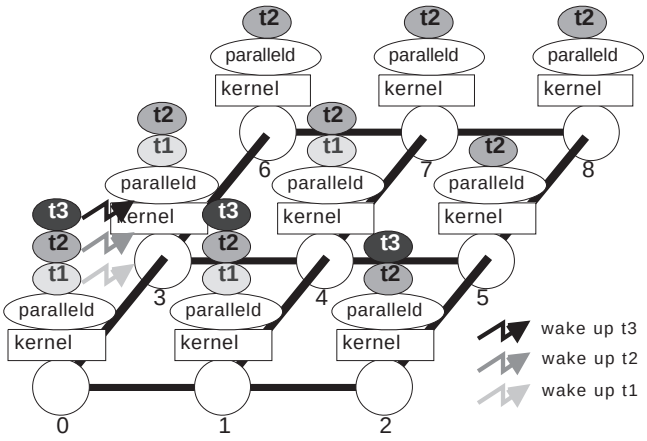

Fig. 7. Process allocation and Synchronization by AP/Linux

cell0 is responsible for the control of wake-up synchronization for all parallel processes.

In Figure 7, three wake-up synchronizations are issued for each process. The wake-up synchronization of parallel processes is controlled by the wake-up of parallel processes on cell0. Just before a parallel process on the scheduling queue of cell0 is activated, the kernel on cell0 informs other cells that they should activate the parallel process, i.e. the kernel on cell0 sends the TID of the parallel process using B-net. The TID is used for identification of parallel process on each cells. The kernels on the cells which have the TID wake up the parallel process at the next scheduling.

The handler for identification and rescheduling of parallel process is implemented by *fast handler*, which is Linux interrupt handler. The fast handler catchs an interrupt just when an interrupt has occurred, but the main handling is done after the current process is suspended. The fast handler only handles the identification of TID just when the wake-up synchronization has occurred. If the cell has the TID, the kernel reschedules after the current process is suspended. It achieves loose synchronization.

Parallel processes can get processing time if the processor is not busy. The scheduling depends on the load on each processor. In Figure 7, while *task3* is running on cell0,1, and 2, *task1* and *task2* may be running on cell3 and 4. However, there is no guarantee of synchronization. The local scheduler decides which processes run.

The process switch sometimes cause problems with messaging on network. To solve this problem, the CM5[14] offers the *all-fall-down* mechanism on network switch. The *all-fall-down* mechanism enqueues messages to the nearest switch when a process switch has occurred. The all-fall-down mechanism clears the

network. The typical overhead of a process switch on the CM5 is reported as 10 ms[18]. The AP1000+ does not offer this facility. On AP/Linux, messages can be alive on the network even after a process switch has occurred. The message header has the TID which is used for identification. Even if the process is switched on the destination cell, the message is identified by the TID and is buffered at the memory space of the process.

5 Design of our TSS on AP/Linux

Unfortunately original AP/Linux cannot make the best use of parallel computers, because of the following reasons.

- No space sharing.
- Wake-up synchronization of parallel process depends on the scheduling queue on cell0.

Original AP/Linux does not allow space sharing. The allocations of parallel processes always start from cell0. These allocations cause concentration of load on cell0.

Wake-up synchronization is issued depending on the scheduling queue on cell0, because the scheduling queue have all PIDs of parallel processes. Only one parallel process can run synchronously at a time. At that time other parallel processes cannot run synchronously. Owing to the wake-up synchronization, space sharing is not implemented.

In order to implement our TSS, allocation of parallel processes should be allowed to start from any cells, namely the allocation must be independent form the kernel on each cell. Furthermore wake-up synchronization of parallel processes should be done by a special facility which is independent from the scheduling queue on cell0, namely, the facility does not implement in a kernel. To settle these requirements, we provide a slice daemon(*sliced*) which takes responsibility of process allocation and wake-up synchronization of parallel processes.

Our TSS design is based on the AP/Linux implementation. We use most of the facilities of the parallel execution environment provided by the original AP/Linux, as we do not want to increase special facilities, and it is easy to implement.

5.1 Process allocation

sliced takes responsibility of space sharing. In the same manner of the original AP/Linux, *prun* requests the *paralleld* on cell0 to execute a parallel process. The new *paralleld* asks the allocation region to *sliced*. The *sliced* searches a processor region using a partitioning algorithm. If a region is decided, *sliced* supplies the region to *paralleld*. Creation of parallel processes are done in the same manner as the original AP/Linux, that is, the parallel process gets the same PID and TID(using *clone()* system call) and is subsequently enqueued in a local scheduling queue.

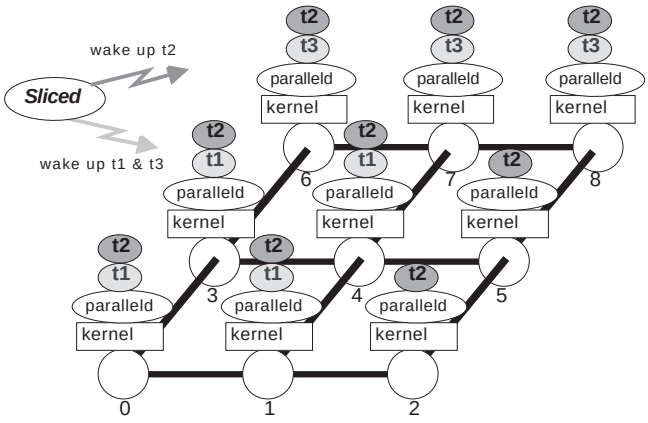

Fig. 8. Process allocation and Synchronization by our method

The partitioning algorithm assumes slices(virtual parallel processors mentioned in Section 2.) and uses the data structure described in Figure 3. Figure 8 shows the allocation required by *prun*s in Figure 6. *task1* and *task2* are allocated on a slice and *task3* is allocated on another slice. *sliced* also manages *multiple tasks* using this data structure.

When a parallel process finishes, it sends a message to *paralleld* in the same manner as the original AP/Linux. Then *paralleld* informs *sliced* of the termination, which deletes the processor region data for the parallel process for reuse. The details of the processor region management are described in the paper[10].

5.2 Wake-up synchronization

The *sliced* holds the data structure which is illustrated in Figure 3, and knows the parallel processes which are on a slice. The parallel processes on a slice are activated simultaneously every certain interval, which are parameterized.

Wake-up synchronization is controlled by *sliced*. In Figure 8, synchronization is issued for *task1* and *task2* and for *task3*. This means that the wake-up of parallel processes is independent from the scheduling queue on cell0.

The synchronization request is issued using B-net, as in the original AP/Linux. The synchronization request issued by *sliced* uses PID instead of TID, because parallel processes are distinguished by PID in *sliced*. A synchronization request can send plural PIDs to achieve low overhead. In Figure 8, a synchronization request is issued to wake up *task1* and *task3*. These cells ,which have the PIDs, handle synchronization requests in the same manner as the original AP/Linux.

6 Current Status

The new *paralleld* and *sliced* are running on the 16 cell AP1000+ at Electrotechincal Laboratory and the Australian National University. Unfortunately it does not offer 2 dimensional partitioning, as this depends on broadcast mechanism of old *paralleld* [1]. It offers 1 dimensional First Fit partitioning. The requirement of rectangular cell region is linearized. For example 2×2 cells request is translated into 4×1 cells and allocated as contiguous cells. This implementation offers multiple tasks as described in Section 2.

The *sliced* sends a wake-up synchronization every $100ms$. The Linux tick time is $16.7(1/60)ms$. Therefore A wake-up synchronization is sent once every 6 process time quantum.

6.1 Simple Performance test

In order to evaluate the effect of our scheduling, we ran test processes and measured elapsed process time.

At first we ran 8 processes which required the similar CPU time and 2 CPUs, namely, matrix multiply. Figure 9 shows the results. The x axis indicates time and the y axis indicates the order of submitted processes, the lowest process(No.1) is submitted first and highest process(No.8) is submitted last. In this figure, lines indicate the start time and finish time of each process. The black lines indicate the results by our method and gray lines indicate the results by the original AP/Linux. The number in front of the parenthesis indicates the required processors and the number in parentheses indicates the allocated processor region.

This result shows the effect of space sharing. The original scheduling allocated all processes on cell 0 and 1. Our method can distribute processes equally on all cells. The 8 processes do not overlap on any cells and run simultaneously. Therefore the theoretical improvement is 1/8.0. However, there is the overhead of synchronization and other processes, including *paralleld* and *sliced*. The practical improvement was 1/7.1 in this case. This result means that the overhead is insignificant, the effect of space sharing could achieve high performance.

The start time of each process(on the left side of the line) depends on the time of allocation. From Figure 9, we know that allocation time of our methods are faster than the original scheduling. The reason for this was that *paralleld* existed on cell0, and all processes were allocated on cell0 by the original scheduling policy. Cell0 had a heavier load. In our method, *paralleld* and *sliced* also exist on cell0 but only a parallel process is allocated on cell0 in this case. The load on cell0 was not high with our method. Therefore the response for allocation could be fast.

Second, we confirmed the effect of the *multiple task*. Figure 10 shows the results with 8 processes, which require random cells(2,6,6,2,2,11,4, and 3). In

[1] The broadcast mechanism is updated[19] and enables to get 2 dimensional regions. We are now updating our *paralleld* and *sliced*.

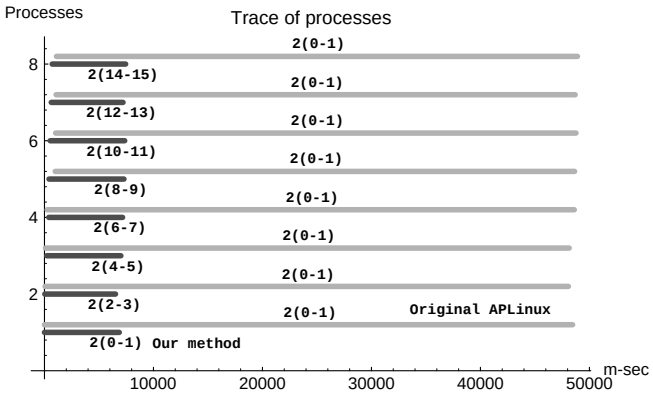

Fig. 9. Trace of parallel processes on original AP/Linux and our method

this case three slices were created. The first slice had three processes(process1,4, and 6 which occupied cell0-1, cell14-15, and cell 2-13 respectively). The second slice had 4 processes(process2,3,4, and 5 which were occupied cell2-7, cell8-13, cell14-15, and cell0-1 respectively). The third slice had 4 processes(process 3,4,7 and 8 which occupied cell8-13, cell14-15, cell0-3 and cell4-6 respectively). Process 3 and 4 could be multiple tasks. Process 3 existed on slice 2 and 3. Process 4 existed on slice 1, 2, and 3. The other tasks were single tasks. In this case process 3 was approximately 2/3 times faster than a single task. Process 4 was approximately 1/3 times faster than a single task. These results indicate that multiple tasks worked well and overhead of allocation and synchronization did not affect processes severely.

In Figure 10 the process 1,5 and 7 showed late finish times. These processes were allocated on cell0. The reason is that cell0 is busy with single processes(*paralleld* and *sliced*) as well as parallel processes. The situation caused unfairness of CPU time for parallel processes. We discuss how to settle this problem in Section 7.

7 Discussions

7.1 Parallel process and UNIX environment

We have implemented scheduling method which combines time sharing and space sharing. The original concepts assume that only parallel processes exist and these parallel processes can switch at the same time. Also it does not consider a process switch caused by virtual memory or I/O. However, in real implementation we must deal with these side effects.

Our implementation deals with the existence of sequential processes on each cell. Some sequential processes are standard UNIX daemons, for example *init*,

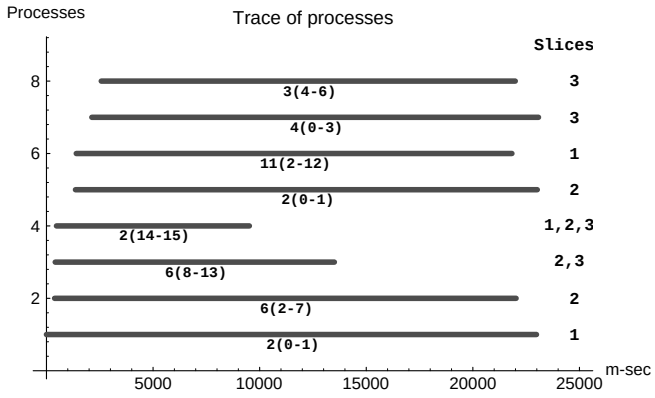

Fig. 10. Effect of multiple tasks

kswap, *kflushd*, and *paralleld*. Parallel processes also use virtual memory resulting in asynchronous pasing events. Both of the effects cause skew of time quantum for each process.

We use autonomous scheduling on each cell to compensate for this skew. When a parallel process on a cell causes virtual memory activity, anther process is activated to compensate for the skew while the parallel process on other cell is running. This is one solution to allow for the UNIX environment still, while running tightly scheduled parallel processes. Unfortunately the combination does not fit nicely in some cases. For example the process switch causes delay in communication with a parallel process. In the original gang scheduling algorithm, there is no process switch in a time quanta. Therefore a parallel process is supposed that there is no interference communication. However, in this implementation the problem arises. We must estimate the effect of the delay and minimize the cost.

7.2 Load distribution

The load of parallel processes can be distributed by our TSS, which uses a partitioning algorithm. However, AP/Linux allows a mix of parallel processes and sequential processes. Unfortunately the *sliced* doesn't observe the load caused by sequential processes. The *sliced* should observe the situation and distribute the load of parallel and sequential processes on each cell. We propose it should act like NQS(Network Queuing System). NQS observes the load on each processor and allocates a job to the processor which has the smallest load. We must consider the new partitioning algorithm for this purpose.

7.3 New partitioning Algorithm

We have proposed a new partitioning algorithm[20] which is a combination of a contiguous and a non-contiguous partitioning algorithm[7,8]. The new algorithm compensates for the weak points of contiguous and non-contiguous partitioning algorithms. It could achieve high processor utilization and quick response. We plan to implement this algorithm on a real machine.

However, there may be difficultiies implementing multiple tasks which exist on some slices, as processor regions of a task are sometimes distributed. This may invoke a significant cross-checking overhead at task load time. The multiple task is useful to decrease the number of slices as well as to increase processor utilization. If we use the non-contiguous partitioning algorithm in our TSS, we must consider the algorithm to check crossing processor regions.

8 Conclusions

In this paper we presented the design of a time sharing system on a real parallel computer, the AP1000+. The time sharing system includes space sharing by a partitioning algorithm, and can make the best use of the number of processors.

The implementation design is based on the modern operating system AP/Linux, because AP/Linux offers many facilities for parallel processing, the parallel execution command *prun*, a daemon for creating parallel processes *paralleld*, and scheduling for parallel processes. Using these facilities, we introduced space sharing on AP/Linux, and separated wake-up synchronization from the scheduling queue on the primary cell. These special facilities for space sharing and wake-up synchronization did not implemented in a kernel. We showed the improvement of the special facilities as a *sliced* daemon for process allocation and synchronization of parallel processes. Performance tests were also taken, showing the effects of space sharing. In the near future, we are planning to measure the exact performance by using some real applications. We will make clear the effect of messaging and scheduling policy.

References

1. P. Chuang and N. Tzeng. An efficient submesh allocation strategy for mesh computer systems. *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 259–263, 1991.
2. K. Li and K. Cheng. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. *Journal of Parallel and Distributed Computing*, (12):79–83, 1991.
3. Y. Zhu. Efficient processor allocation strategies for mesh-connected parallel computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.
4. J. Ding and L. N. Bhuyan. An adaptive submesh allocation strategy for two-dimensional mesh connected systems. *Proceedings of International Conference on Parallel Processing*, pages (II)193–200, 1993.

5. D. D. Sharma and D. K. Pradhan. A fast and efficient strategy for submesh allocation in mesh-connected parallel computers. *Procedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 682–689, 1993.
6. S.M. Yoo and H.Y. Youn. An efficient task allocation scheme for two-dimensional mesh-connected systems. *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 501–508, 1995.
7. W. Liu, V. Lo, K. Windish, and B Nitzberg. Non-contiguous Processor Allocation Algorithms for Distributed Memory Multicomputers. *Supercomputing*, pages 227–236, 1994.
8. V. Lo, K. Windish, W. Liu, and B Nitzberg. Non-contiguous Processor Allocation Algorithms for Mesh-connected Multicomputers. *IEEE Trans. on PARALLEL AND DISTRIBUTED SYSTEMS*, 8(7):712–726, 1997.
9. B. Yoo, C. Das, and C. Yu. Processor management techniques for mesh-connected multiprocessors. *Proceedings on International Conference on Parallel Processing*, pages II–105–112, 1995.
10. K. Suzaki, H. Tanuma, S. Hirano, Y. Ichisugi, and M. Tukamoto. Time sharing systems that use a partitioning algorithm on mesh-connected parallel computers. *The Ninth International Conference on Parallel and Distributed Computing Systems*, pages 268–275, 1996.
11. J.K. Ousterout. Scheduling techniques for concurrent Systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.
12. D. Feitelson and L. Rudolph. Distributed Hierachical Control for Parallel Processing. *IEEE COMPUTER*, 23(5):65–77, 1990.
13. A. Tridgell, P. Mackerras, D. Sitsky, and D. Walsh. Ap/linux a modern os for the ap1000+. *The 6th Parallel Computing Workshop*, pages P2C1–P2C9, 1996.
14. *Connection Machine CM-5 Technical Summary.* Thinking Machines, 1992.
15. D. Feitelson. Packing Schemes for Gang Scheduling. *Lecture Notes in Computer Science 1162*, pages 89–110, 1996.
16. A. Hori, Y. Ishikawa, H. Konaka, M. Maeda, and T. Tomokiyo. A scalable time-sharing scheduling for partitionalble, distributed memory parallel machines. *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 173–182, 1995.
17. D. Sitsky, P. Mackerras, A. Tridgell, and D. Walsh. Implementing MPI under AP/Linux. *Second MPI Developers Conference*, pages 32–39, 1996.
18. D. C. Burger, R. S. Hyder, B. P. Miller, and D.A. Wood. Paging Trade off in Distributed Shared-Memory Multiprocessors. *Supercomputing*, pages 590–599, 1994.
19. D. Walsh. Parallel process management on the ap1000+ under ap/linux. *The 7th Parallel Computing Workshop*, pages P1V1–P1v5, 1997.
20. K. Suzaki, H. Tanuma, S. Hirano, Y. Ichisugi, C. Connelly, and M. Tukamoto. Multi-tasking method on parallel computers which combines a contiguous and a non-contiguous processor partitioning algorithm. *Lecture Notes in Computer Science 1184*, pages 641–650, 1996.