7. 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.
8. A. Hori, H. Tezuka, and Y. Ishikawa. Overhead analysis of preemptive gang scheduling. *IPPS'98 Workshop on Jpb Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science 1125)*, pages 217–230, 1998.
9. P.G. Sobalvarro and W.E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. *IPPS'95 Workshop on Jpb Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science 949)*, pages 106–126, 1995.
10. P.G. Sobalvarro, S. Pakin, W.E. Weihl, and A.A. Chien. Dynamic coscheduling on workstation clusters. *IPPS'98 Workshop on Jpb Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science 1125)*, pages 231–256, 1998.

computers. Scheduling here is in principle moderate co-scheduling, with which the order of priority of the specific parallel processes is controlled per fixed time. In addition, as a method to change the order of priority of parallel processes simultaneously for conducting co-scheduling, the authors proposed and installed internal synchronization which takes advantage of the synchronous clock that is a hardware property of AP1000+.

Together with the communication library for polling/signal switching, this method allowed efficient execution of both fine grain parallel processes and coarse grain parallel processes. It was confirmed that co-scheduling allows efficient operation of busy wait communications of fine grain parallel processes, which wait for communication by busy wait, even when several of those are input. In addition, it was confirmed that the overall processing efficiency of coarse grain parallel processes, which wait for communication by signal, could be improved by issuing context switching to execute another parallel process.

Although the problem with internal synchronization is the co-scheduling skew caused by the failure of synchronization in the respective processors when conducting re-scheduling, it was concluded that the time lag was negligible since it accounted for around 2.5% of 200 ms where priority switching takes place.

The important issue of a parallel OS is to establish both a communication library and a scheduler in consideration of the properties of applications and the structures of the hardware available. A parallel OS designer must decide whether to use waiting by busy wait or context switching based on the communication patterns of the applications. If co-scheduling can efficiently be installed on usable hardware, it is sufficient to consider its introduction for fine grain parallel processes. At present, AP/Linux is compatible with both fine grain and coarse grain parallel processes, thus allowing efficient processing of a wide variety of application programs.

References

1. 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.
2. J.K. Ousterout. Scheduling techniques for concurrent Systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.
3. A.C. Dusseau, R.H. Arpaci, and D.E. Culler. Effective Distributed Scheduling of Parallel Workloads. *SIGMETRICS'96*, 1996.
4. D. Sitsky, P. Mackerras, A. Tridgell, and D. Walsh. Implementing MPI under AP/Linux. *Second MPI Developers Conference*, pages 32–39, 1996.
5. D.G. Feitelson. *Job Scheduling in Multiprogrammed Parallel Systems.* IBM research Report RC 19790(87657), 1997.
6. 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.

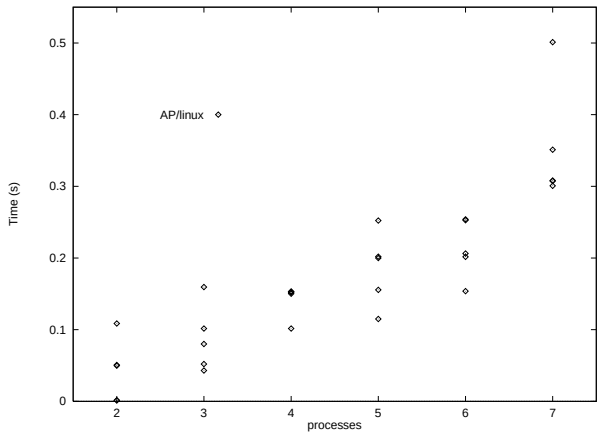


Fig. 9. Speed of parallel process creation.

On the other hand, the time required for processing by interruption in the respective kernels was about 0.3 ms for 2 processes and about 0.4 ms for 8 processes. Although they are added to parallel processing as overhead every time, they are considered negligible because they account for 0.4% of 200 ms. This is also considered negligible when compared with the time lag of synchronization.

5 Related Research

Implicit co-scheduling[3] and dynamic co-scheduling[9, 10] are resemble to our moderate co-scheduling. Implicit co-scheduling and dynamic co-scheduling are demand-based co-scheduling, that is, they relay on a local scheduler to run the parallel process when it receive a message. They suppose that the scheduler quickly dispatches parallel processes after receiving a message. We believes our local scheduler cannot dispatch so quickly. Therefore we offered multi-processor-wide scheduling. The multi-processor-wide co-scheduling is used to synchronize the scheduling of parallel processes on all processors. It is beneficial for fine grain parallel processes, because it raises the probability of running suitable parallel processes on the allocated processors at the same time. It causes small loss time of message receive for frequent messages. Furthermore moderate co-scheduling, unlike gang scheduling, allows to yield process time when a parallel process must wait long time. It caused high through-put of parallel processes.

6 Conclusions

In this paper, the authors proposed a method of scheduling, with which space sharing and time sharing are combined, for AP/Linux which is an OS for parallel

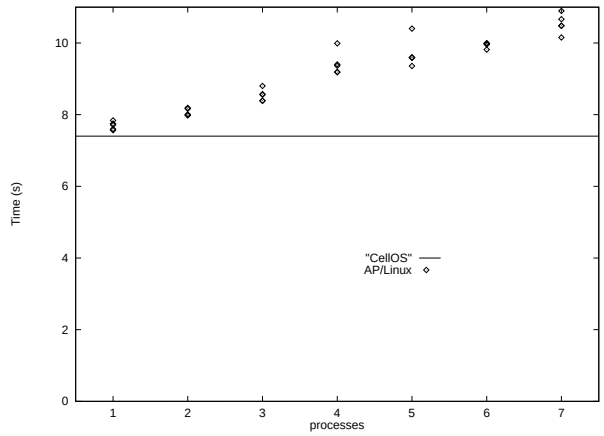


Fig. 8. Elapse time of round robin which transfer 2000 byte.

because other processes cannot be processed during the time allocated to one parallel process.

Figure 9 shows the speed of creating several parallel processes. This figure shows the difference between the time when the first parallel process is started and the time when the last parallel process is started, when several parallel processes are input. The difference in the starting time increased almost linearly as the number of input processes increased. Process creation is subject to the influence of local scheduling because it is dependent upon the scheduling of paralleld. If the timing of scheduling a parallel process is delayed, another parallel process is executed. Since the parallel process that missed the chance must wait until its paralleld becomes the next target of scheduling, the difference in the starting time is great in some cases. There are cases where the time for creating two or three processes is almost equivalent, which is because paralleld received the demand of parallel process creation consecutively during execution. Figure 9 shows that the speed of parallel process creation is not severely affected, thus proving that the time is appropriate for a scheduling method in conformity to the style of UNIX.

4.3 Performance of Synchronization

The co-scheduling skew by internal synchronization averaged about 10 ms with the standard deviation of around 5 ms. Although this average of co-scheduling skew appears to be great for the switching of 200 ms, it does not affect the scheduling itself greatly because it is more similar to a time shift rather than skew. Although the problem with the skew is rather the standard deviation, it is considered allowable as a co-scheduling skew, since 5 ms accounts for 2.5% of 200 ms where switching of priority takes place.

cesses on the host computer can be overlapped with the execution of parallel processes. This shows that the parallel processes on other slices are not obstructed even when there are several slices. This result also shows that the influence of co-scheduling is small. The performance of co-scheduling will be described in Section 4.3.

Table 3. Result of one to all.

(2×2)				(4×2)			
slice	process	sec	times	slice	process	sec	times
1	1	2.78	1.00	1	1	5.05	1.00
	2	2.95	1.06		2	5.28	1.05
	3	3.13	1.13	2	3	9.73	1.93
	4	3.38	1.22		4	9.85	1.95
2	5	5.49	1.97	3	5	14.34	2.84
	6	5.63	2.03		6	14.47	2.87
	7	5.86	2.11	4	7	19.11	3.78

Effect of Moderate Co-Scheduling With AP/Linux, the time allocated to parallel processes is not strictly secured, unlike the case with gang scheduling. Accordingly, when the parallel process being executed becomes I/O waiting, it is possible to move to another process by context switching. To confirm this, the execution of a coarse grain parallel process, with which the state is switched to signal waiting without waiting by busy wait, was studied. In the environment where several parallel processes are running, processing of one parallel process is likely to be overlapped with that of other parallel processes.

As an example of coarse grain, a parallel process that makes a message of 2,000 bytes travel 1,000 times around all cells (8×2) by round robin was used. Figure 8 shows the results of the total processing time. For reference, the results of execution using CellOS are also shown in the figure. Here, the overhead required for starting CellOS is excluded. With CellOS, communication is waited by busy wait. When the round robin of 2,000 bytes is one process, the processing time of waiting by busy wait was almost equivalent to that of waiting by switching over to signal.

Although the effect of moderate co-scheduling is not apparent when only a single parallel process is executed, the effect is evident when several parallel processes are executed. Despite the overhead of context switching to signal waiting, the entire processing time is not greatly increased since other processes can be executed meanwhile. Figure 8 shows that the total processing time does not increase greatly because of the overlapped execution of several parallel processes. With strict gang scheduling, the effect of such overlapping cannot be gained

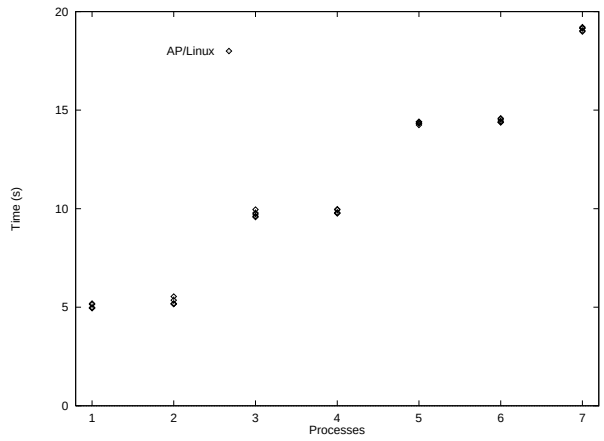


Fig. 7. Effect of space sharing and time sharing for one to all (4×2).

Effect of Space Sharing/Time Sharing Several fine grain parallel processes as described before were input by giving variations to their cell realms. Only the case with "one to all" is discussed here, since all the parallel processes executed by "pipelined round robin" and "one to all" presented the same trend. Figure 6 and Figure 7 shows the results of execution in two graphs of demanded cell realms, one for 2×2 and the other for 4×2. In Figure 6 and Figure 7 , the X axis represents the number of processes, while the Y axis shows the total processing time. Parallel processes were executed 5 times for each to confirm the operation to be stable. Both of these graphs prove that the space sharing and time sharing are effective. Parallel processes that demand the cell realm of 2×2 can be laid out spatially by 4 parallel processes per slice, thus increasing the total processing time in units of 4. In the same manner, the total processing time of parallel processes that demand the cell realm of 4×2 increases in units of 2.

Table 3 tabulates the values of Figure 6 and Figure 7. These tables show the state of slice allocation per number of process, the mean average of the total processing time required, and the relative elongation based on total processing time per process. It is known from these tables that the total processing time increases only slightly when the number of parallel processes that can be executed concurrently is increased by space sharing without changing the number of slices. The increase in the total processing time is attributable to the delay in creating processes and the slight elongation of the time required for processing the respective parallel processes. However, since this increase in time is slight in comparison with the time required for processing parallel processes, it is negligible and can be offset by the effect of space sharing.

The relative elongation of the total processing time is shown to be proportional to the increase of slices. This is super linear in the results of 4×2 (Table 3), which is due to the fact that the processing of the management of parallel pro-

preprocessing for starting the parallel process and post processing. The values of the total processing time are almost the same regardless of the frequency of the iteration of data transmission, although this may not be clear in the figure because the values are shown using the log scale. Table 2 summarizes the time required for the overhead, which was about 5.4 seconds with CellOS and about 0.8 seconds with AP/Linux. The difference is attributable to the difference in the method of running parallel processes. With CellOS, both the execution format code of the parallel process and CellOS itself are transmitted to the respective cells, when starting parallel processing, and the respective states need to be initialized for executing a parallel process. With AP/Linux, since the execution format code of the parallel process is managed by demand paging, it is not loaded from the disk if already loaded on the memory. Since the processing was conducted several times in the measurement, the time required for loading the code from the disk was eliminated. In addition, since the parallel process of AP/Linux can use the dynamic library, there is no need to load the code itself although there is overhead of dynamic link. AP/Linux, which is equipped with these functions of UNIX, is advantageous when executing several parallel processes.

4.2 Processing of Several Parallel Processes

The effects of space sharing/time sharing when executing several fine grain parallel processes and the effect of moderate co-scheduling when executing coarse grain parallel processes are shown below.

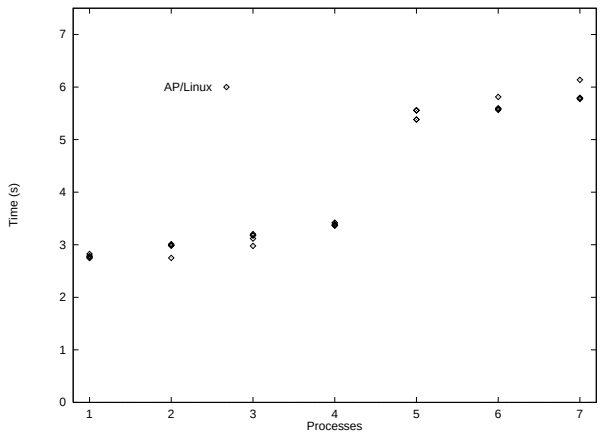
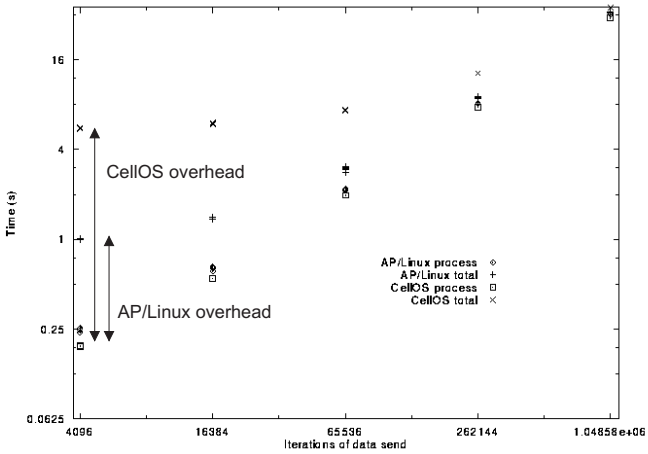
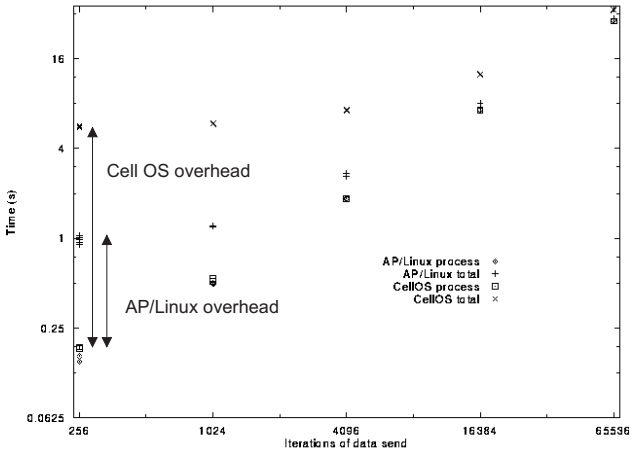


Fig. 6. Effect of space sharing and time sharing for one to all (2×2) .



pipelined round robin



one to all

Fig. 5. Comparison CellOS and AP/Linux.

A program with message patterns of "pipelined round robin" and "one to all" was adopted for processing fine grain parallel processes for performance evaluation. "Pipelined round robin" transmits data one after the other just like a pipeline to travel around all cells (8×2). "One to all" sends a message from one cell to all other cells (8×2)-1, and the receivers of the message return answerbacks without any particular processing. Both of them conduct data transmission of 10 bytes.

Figure 5 shows the results of executing a parallel process by both "pipelined round robin" and "one to all" using CellOS and AP/Linux, respectively. The parallel process was executed five times for each to confirm the operation to be stable. In Figure 5, the X axis represents the frequency of the iteration of data transmission, while the Y axis is the time. Since the frequency of the iteration of data transmission was checked at every power of 4, both the X axis and Y axis are represented using the log scale. Figure 5 also shows the time required from the commencement of input of the process from the host computer until the termination of processing (total processing time, which is indicated as "total" in the figure) and the time required from the commencement of actual processing on AP1000+ until termination (process processing time, which is indicated as "process" in the figure). The difference between the total processing time and the process processing time is the overhead between preprocessing and post processing by OS. The overhead of co-scheduling by AP/Linux is included in the process time.

Figure 5 shows that there is no major difference between CellOS and AP/Linux in regard to the process processing time required by the parallel process itself. The same applies to "pipelined round robin" and "one to all"; there was no difference in the performance due to communication patterns. Although the overhead of co-scheduling is included in the process processing time of AP/Linux, it is within the negligible range in comparison with the execution by CellOS. Thus, these results show that the execution performance does not change greatly even if AP/Linux is used instead of CellOS. AP/Linux is even superior to CellOS in terms of usability, considering the overhead that arises when starting CellOS and the capability of AP/Linux to execute several processes concurrently.

Table 2. Overhead to run a parallel process.

pipelined round robin		one to all	
	Average (sec)		Average (sec)
CellOS	5.38	CellOS	5.39
AP/Linux	0.83	AP/Linux	0.77

The total processing time from the input of the parallel process from the host computer until the termination of processing differs greatly between CellOS and AP/Linux. This is attributable to the difference in the overhead between

Every time re-scheduling is conducted, the system checks whether or not the time for changing the priority of parallel processes has elapsed in the respective cells. If the time is already over, the priority of the parallel process that has been executed is lowered, and the priority of the next parallel process to be executed will be returned to normal.

When the processing of a parallel process is terminated, the process sends a message of termination to paralleld. Upon receipt of this message, paralleld disconnects the standard input/output session between prun and the parallel process, then sends a message of parallel process termination to pds. Upon receipt of this message of termination, pds notifies it to bootap+. Bootap+ releases the realm of the parallel process, and notifies the slice information, that has been renewed by interruption, to the kernels of the respective cells. Based on this information, the kernels renew the slice queue information.

The time intervals for switching the priority are set to be sufficiently longer than the time required for re-scheduling. If re-scheduling fails to operate for some reason, a parallel process that should be executed at this point will be calculated from the slice queue to switch over to the processing of the appropriate parallel process.

Problems There is a problem with internal synchronization, i.e., co-scheduling skew may occur because re-scheduling does not occur simultaneously in the respective processors. Nevertheless, the influence of the co-scheduling skew is considered to be small because the only processes that require processing are parallel processes except for daemon and because parallel processes are controlled by priority switching. The extent of the co-scheduling skew will be identified through actual installation.

4 Performance Evaluation

To evaluate the installed scheduling, parallel processes were input for measuring the performance of execution. AP/1000+ of 16 cells (8×2) was used for the performance evaluation, and the priority of parallel processes was set to be switched over at every 200 ms. The period of co-scheduling was reasonable to compare to Score-D[8] (the period was 50ms-200ms, CPU was 200Mhz PentiumPro) and the period was enough to hide the overhead as mentioned in section 4.3.

The focus of the measurements was to compare AP/Linux with CellOS, and to identify the efficiency when executing several fine grain parallel processes that wait for communication in the busy wait mode, and coarse grain parallel processes that wait for communication in the signal mode.

Unfortunately we did not have suitable benchmark programs to compare fine and coarse grain parallel process. So we offered original program to check the performance.

4.1 Comparison with CellOS

First of all, one parallel process was executed to compare AP/Linux with CellOS.

single tasks, while rectangles colored in thick gray represent multiple tasks. To improve response, slices without single task were eliminated.

Bootap+ is equipped with the function of parallel process layout. Since bootap+ occupies BIF when AP/Linux is in operation, bootap+ needs to be equipped with a function of processing by interruption.

AP1000+ is a two-dimensional torus link. With AP/Linux, however, it is supposed to be a mesh link, for which many partitioning algorithms are proposed. For the partitioning algorithms, Adaptive Scan[7] was adopted because of its high efficiency.

3.3 Synchronization

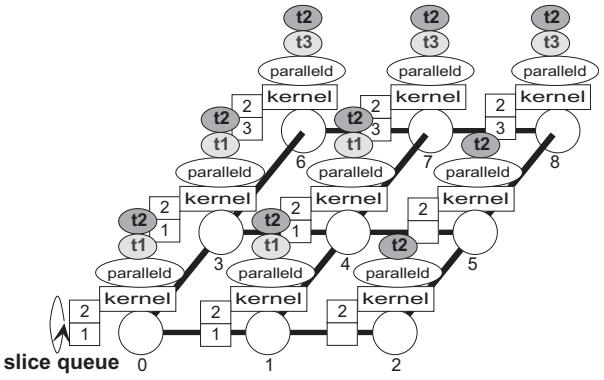


Fig. 4. Internal synchronization.

For simultaneously controlling the priority of parallel processes on slices, the authors propose a method of internal synchronization that takes advantage of the hardware properties of AP1000+. With internal synchronization, a clock, with which time synchronization is physically guaranteed, is used. Since the clock ticks inside the respective cells are operated at 80 ns, synchronization can be conducted without relying upon the host computer.

Figure 4 shows the schematic diagram of internal synchronization. With internal synchronization, slice queues with slice information are inside kernels in addition to ordinary process queues. With internal synchronization, parallel processes are laid out by bootap+ of the host computer. Every time layout is conducted, bootap+ interrupts the respective cells to renew the slice queue information.

Regarding time adjustment of co-scheduling, Cell0 becomes the representative and notifies the present time to all cells by interruption when slices are created. This time is the standard time, and the priority of parallel processes is switched over every time the time allocated to each parallel process has elapsed.

through paralleld.

When a parallel process is terminated, the process transmits a message of termination to paralleld. Upon receipt of this, paralleld disconnects the standard input/output session between prun and the parallel process, then sends a message of parallel process termination to pds. Upon receipt of this message, pds notifies the termination to bootap+, and releases the realm to which the parallel process was allocated.

3.2 Layout of Parallel Processes

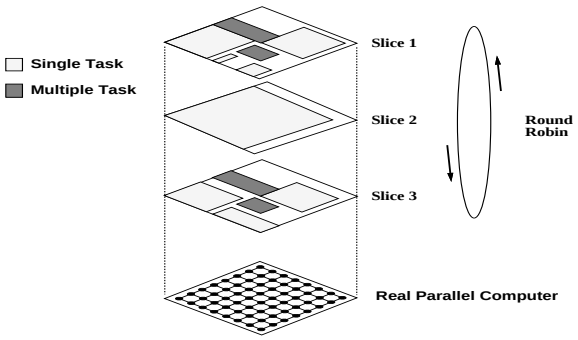


Fig. 3. Combination of time sharing and space sharing.

The layout of parallel processes is managed by a method proposed by the authors[6], which is a combination of partitioning and virtual parallel computers. Virtual parallel computers with a linkage system identical to an actual parallel computer are prepared for time sharing, while space sharing is conducted on the respective virtual parallel computers using partitioning algorithms. Such a virtual parallel computer is called a slice. Extra slices are prepared for allocating the excessive parallel processes that cannot be allocated to existing slices. When conducting time sharing, actual parallel computers are allocated to these slices per certain time by round robin. The efficiency in using the processor space was improved by such time sharing, with which one slice was prepared for a parallel process that uses the entire group of processors, and other slices were allocated to other parallel processes that do not require processors much.

Figure 3 shows the state of slices in time sharing. Rectangles on the respective slices represent the allocation of parallel processes. The rate of processor utilization in one slice can be raised by partitioning algorithms. Parallel processes that can be allocated to several slices exist in the slices, and can increase the rate of processor utilization. A parallel process that exists in only one slice is called a single task. A parallel process that can exist in more than one slice is called a multiple task. In Figure 3, the rectangles colored in thin gray represent

Table 1. Command and servers for parallel process creation and management.

name	type	function	machine
prun	command	parallel process request	any machine
pds	server	prun & paralleld management	host
bootap+	server	allocation management	host
paralleld	server	create parallel process	each cell
kernel	interrupt	priority control	each cell

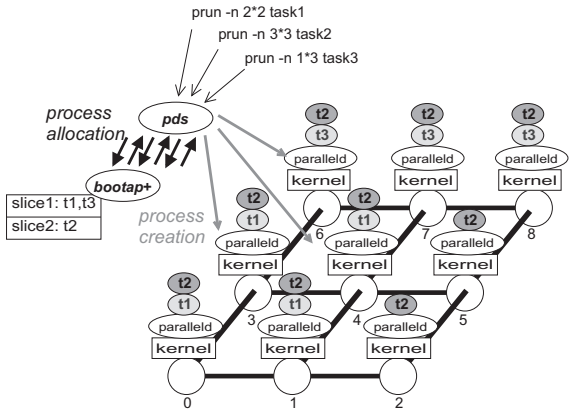


Fig. 2. Parallel process creation.

The prun command requests pds (parallel daemon server) on the host computer to secure and create cells. Pds checks the appropriateness of both the requested cell size and the parallel processes. If they are judged appropriate, pds requests cell layout to bootap+. Upon determination of the locations of layout by bootap+, pds requests paralleld on the cells, where the parallel processes are scheduled to be laid out, to create parallel processes. To create parallel processes, paralleld uses the revised version of the clone function that is in the standard package of Linux. With this clone function, the same process ID (PID) can be designated on the respective cells from outside of kernels, and parallel processes can be discriminated among cells. The PID of the parallel process utilizes the upper half of the entire PID, while that of the sequential processes utilizes the lower half. Pds manages the PID of parallel processes. Parallel processes with designated PID are created on the respective cells, and are put into the respective local process queues.

Paralleld relays the standard input and output between prun and parallel processes. Input from the terminal which is executing prun is transmitted through paralleld, and output from parallel processes is also transmitted to the terminal

processors. When conducting co-scheduling, the order of priority is controlled by giving an ordinary order of priority only to the parallel process that is the target of execution, while lowering the priority of other parallel processes which are not the targets of execution. At present, the values of the order of priority of the parallel processes that are not the targets of execution are determined by deducting 15 from their respective values of priority. The priority of parallel processes is not raised to avoid impeding the processing of daemon that is indispensable for UNIX processing and daemon (paralleld) that creates parallel processes. When the target parallel process does not become executable by the local scheduler, another parallel process becomes executable instead. This method is exclusively for managing the priority, and does not guarantee that a parallel process is executed infallibly immediately after co-scheduling. If there is some other process with priority higher than that of parallel processes, the local scheduler will give priority to the former.

Although process thrashing may occur, this can be avoided because parallel processes are the only processes that require a long CPU time, and they are under priority control. Although daemon processing of sequential processes is required in case of emergency, it scarcely affects the parallel processes because the time required for the daemon processing is short. In addition, the throughput can be improved by giving priority to daemon processing. In other words, since the processing of "paralleld," which creates parallel processes and exists in each processor as daemon, is not hindered by a running parallel process, parallel processes can be created promptly. If the processing is hindered by a parallel process, the throughput cannot be improved because it takes time to create new parallel processes. Furthermore, when processing several coarse grain parallel processes at the same time, which are large-scale but have low communication frequency, other parallel processes can be made executable by switching over to the signal waiting mode and activating context switching, by which the throughput can also be improved.

The method of simultaneously changing the order of priority for co-scheduling is described in detail in Section 3.3.

3.1 Creation of Parallel Processes

Parallel processes are created and managed by the command and servers as shown in Table 1.

Figure 2 shows an example of executing a parallel program that is made by using the AP library and the MPI library by issuing the prun command. The circles in the figure represent cells, and both kernel and paralleld exist in the respective cells. By the -n option, the prun command designates the required cell size and the target parallel process to be executed. The prun command can be issued from both the host computer and other computers that are communicable by socket. This figure shows a case example of a demand of creating three parallel processes. Task1 demands the execution of creation in the cell group of 2×2, while task2 and task3 demand it in the cell groups of 3×3 and 1×3, respectively.

By the bootap+ program, AP/Linux loads the kernel onto each cell from the host computer through BIF that is connected to B-net, and mounts disks in each cell or the disk on the host computer onto the file system. When inetd daemon is activated on each cell, logging-in from external computers becomes feasible. The ordinary Linux environment is provided when logged into cells. Since bootap+ provides a virtual console session to each kernel, it cannot terminate a process while AP/Linux is active. Meanwhile, the environment can also be used as an ordinary decentralized environment, since TCP/IP can be used on B-net.

As environments for parallel programming, both the MPI library[4] and the AP library, the latter of which is compatible with a library exclusively provided by AP1000+, can be used. They are communication libraries, with which T-net can be used on the user level. Transmission mistakes of messages sent by such communication libraries are unlikely, since messages are tagged, and destination processes can be judged after they are stored in the ring buffer in the cells of the receivers. Thus, context switching of parallel processes will not affect such messages on the network.

A polling/signal system is installed on these libraries for detecting messages, with which the throughput is improved by issuing context switching in the signal waiting mode, after waiting for a certain period of time by polling (busy wait). When the waiting time is short, however, it is better to wait by polling without issuing context switching, because it will expedite communications and help avoid frequent context switching (processor thrashing). To conduct polling efficiently, all parallel processes need to be activated simultaneously. To meet this condition, it is required to issue context switching by co-scheduling the parallel processes that are dispersed in the respective cells.

The original AP/Linux has a simple parallel process scheduler. Since this scheduler does not provide space sharing, parallel processes are laid out from Cell0 without variation, which causes load concentration. In addition, co-scheduling of parallel processes is dependent upon the local scheduler of Cell0 that has the process IDs of all parallel processes. Specifically, with this scheduler, Cell0 is subject to a huge load, and space sharing is infeasible.

3 Parallel Process Scheduling of AP/Linux

In this paper, the authors propose a scheduling method to be installed on AP/Linux, which provides scheduling by combining space sharing and time sharing.

Time sharing is entrusted to the server of the host computer. To run parallel processes, a message is sent to the server of the host computer, and the server determines the cell realm where parallel processes are executed. The creation and layout management of parallel processes are described in detail in Sections 3.1 and 3.2.

Preempting for time sharing is conducted by using the Linux kernel function on the respective cells. Parallel processes can be co-scheduled by controlling the order of priority of the parallel processes in the local scheduler in the respective

and problems of this method in installation will be identified later.

Hereafter, the outline of AP/Linux will be explained in Section 2; the scheduler to be installed on AP/Linux and its method will be described in Section 3; the results of performance evaluation conducted by running actual parallel processes will be presented in Section 4; a comparison of these results with other relevant research will be shown in Section 5; and conclusions will be discussed in Section 6.

2 Outline of AP/Linux

AP/Linux[1] is a parallel operating system for AP1000+ parallel computers. AP1000+ provides an original operating system called CellOS. When executing parallel processes, CellOS is loaded along with execution format codes and provides only the single user single process environment which does not reside permanently on the processor of AP1000+. AP/Linux is a parallel operating system developed by the CAP group of the Australian National University to overcome this fault.

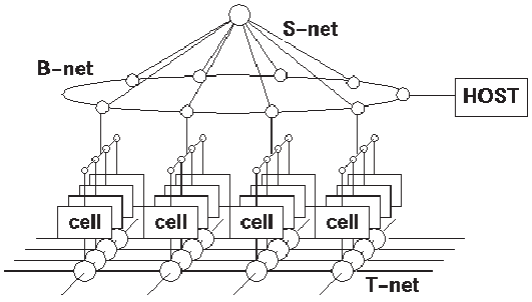


Fig. 1. Overview of AP1000+.

Figure 1 shows the overview of AP1000+. With this computer, the unit of processing elements is called the cell, and each cell has a SuperSparc which runs at 50 MHz. As the interface with the outside, BIF (Broadcast InterFace) is connected to SBUS of the host computer (Sparc Station). Between cells and on the network connected to the host computer are T-net, B-net, and S-net. T-net, which is a two-dimensional torus network, connects adjacent cells with an inter-cell linkage bandwidth of 25 MB/s, and provides worm hole routing. For inter-cell communications, T-net provides message transmission (send, receive) and remote memory access (put, get). B-net, which is a broadcast network with a band of 50 MB/s, connects all cells to the host computer. S-net, which is a synchronous network, connects all cells to the host computer.

proposed is combined scheduling, with which space sharing and time sharing are combined to make the most of their respective advantages. AP/Linux adopts the combined scheduling because of its high efficiency.

When preempting a parallel process by time sharing scheduling, the communication properties of the parallel program to be executed must be taken into account. In the case of fine grain parallel processes that conduct small-scale communications frequently, it is considered more efficient to schedule parallel processes of the respective processors to be scheduled simultaneously (co-scheduling[2]) and wait for messages by "busy wait." On the other hand, in the case of coarse grain parallel processes that conduct large-scale communications less frequently, it is considered better to improve the throughput by context switching, since it takes a long time to wait for messages by "busy wait." There is a communication method that deals with both fine grain and coarse grain parallel processes appropriately, with which context switching is conducted after waiting for messages several times by "busy wait"[3]. The communication library of AP/Linux is equipped with this method, and its effect has already been confirmed [4]. Even if this method is adopted, however, it is still important in the case of fine grain parallel processes to conduct co-scheduling for using the "busy wait" effectively.

A method for co-scheduling all parallel processes simultaneously is gang scheduling[5], with which a certain CPU time is allocated to parallel processes by starting them simultaneously, and execution of other processes is prohibited to guarantee busy wait communications. However, since strict gang scheduling blocks other processes, it is not possible to switch over to another process even when daemon processing is urgently required or when blocking occurs during I/O processing on one of the processors. In addition, when issuing context switching of parallel processes, gang scheduling may require an save/restore mechanism to avoid crashes and delivery mistakes of communication messages on the network.

In this paper, the authors propose a method for conducting co-scheduling while relaxing the strict conditions of gang scheduling and controlling the order of priority of parallel processes managed by the local scheduler in the respective processors. Although this method gives priority of processing to one selected parallel process, it does not impede the processing of other processes that need to be processed urgently due to daemon or other reasons. In addition, when the target parallel process is not executable due to I/O waiting or other reasons, another parallel process can be processed. By actually installing this scheduling method, the authors will show its superiority in terms of overall efficiency, despite a slight sacrifice of communication queuing. In contrast to strict gang scheduling, this method is called moderate co-scheduling.

The mechanism to change the priority of parallel processes simultaneously is the central issue in installation. AP/Linux is equipped with internal synchronization that uses a synchronized clock of the respective processing elements, taking advantage of the hardware properties of AP1000+. With internal synchronization, the accountable time of the priority of parallel processes is investigated every time the local scheduler conducts re-scheduling, and the priority is given to another parallel process when the accountable time is over. The performance

Scheduling on AP/Linux for Fine and Coarse Grain Parallel Processes

Kuniyasu Suzaki[1], David Walsh[2]

[1] Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, 305 Japan
suzaki@etl.go.jp
[2] Australian National University
Canberra, ACT 0200 Australia
dwalsh@anu.edu.au

Abstract. This paper presents a parallel process scheduling method for the AP/Linux parallel operating system. This method relies on 2 schedulings; local scheduling on each processor and global scheduling which is called moderate co-scheduling. Moderate co-scheduling schedules simultaneously parallel processes on each processor by controlling priorities of parallel processes. This method differs from gang scheduling in that it does not promise the running of a parallel process on all processors at the same time. Moderate co-scheduling only suggests a suitable current process to the local scheduling. However, this is good solution for fine and coarse grain parallel processes, because Moderate co-scheduling tells the timing to schedule simultaneously for fine grain parallel processes (tightly-coupled processes on each processor, which requires quick and frequent communication), and local scheduling can yield CPU time when coarse grain parallel processes (loosely-coupled processes on each processor, which cause long wait and less frequent communication) must wait for long time. The method is implemented using AP1000+ special hardware. We call the implementation "Internal synchronization" which uses the synchronized clock. The co-scheduling skew of the implementation was about 2% in the period of moderate co-scheduling was 200ms.

1 Introduction

This paper describes the installation and performance of scheduling for efficient execution of parallel processes on a parallel OS called AP/Linux[1] that is designed for AP1000+ parallel computers. Research on parallel process scheduling has widely been conducted for efficiently operating parallel computers and providing responsive service to individual users. One approach to achieve these is "space sharing" scheduling, with which several parallel processes are laid out efficiently in the space of processors. Space sharing scheduling has been conducted actively since the beginning of the 1990's in line with research on regional management (partitioning algorithms) that fits the respective network topologies. Another approach is "time sharing" scheduling, with which responsivity is improved by preempting parallel processes. One more approach that has been