Overhead Analysis of Preemptive Gang Scheduling

Atsushi Hori[1], Hiroshi Tezuka[1], Yutaka Ishikawa[1]

Tsukuba Research Center
Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305-0032, JAPAN
TEL:+81-298-53-1661, FAX:+81-298-53-1652
{hori,tezuka,ishikawa}@rwcp.or.jp

Abstract. A preemptive gang scheduler is developed and evaluated. The gang scheduler, called SCore-D, is implemented on top of a UNIX operating system and runs on workstation and PC clusters connected by Myrinet, a giga-bit class, high-performance network.
To have high-performance communication at the user-level and a multi-user environment simultaneously, we propose *network preemption* to save and restore network context as well as process contexts when switching distributed processes. We also developed a high-performance, user-level communication library, PM. PM and SCore-D collaborate for the network preemption. When user processes are gang-scheduled, communication messages are first flushed, then the messages and pending messages in the receive and send buffers are saved and restored. Unlike CM-5's All-Fall-Down mechanism, our gang-scheduling scheme is all software; no special hardware support is assumed. Also there is no limitation on network topology and partitioning.
The overhead of the gang scheduler is measured on our new PC cluster, which consists of 64 PentiumPros connected by Myrinet. NAS parallel benchmark programs are used for the evaluation. We found that the message flushing time and network preemption time depends on the communication patterns of the application programs. We also found that the time of saving and restoring network context occupies more than two third of gang scheduling time. Evaluation shows that the slowdown of user program execution due to the gang scheduling is less than 9 %when the time slice is $100 msec$.

1 Introduction

Gang scheduling is efficient for the scheduling of frequently communicating processes[Ous82,GTU91,FR92]. Gang scheduling also enables time sharing scheduling, which provides shorter response times and interactive parallel programming. However, despite the benefits of gang scheduling, there have been few implementations (Table 1).

Table 1. Gang schedulers on distributed memory parallel machines

Scheduler	Platform	Comm. Level	Pre-emptive	Hardware Support
(anonymous)[FR92]	Makbilian	OS	Yes	Yes
CMOST[Thi92]	TMC CM-5	User	Yes	Yes
Medusa[OSS80,Ous82]	Cm*	OS	Yes	Yes
Meiko CS-2	Meiko CS-2	OS	Yes	N/A
MPCI GangScheduler[GW95]	BBN TC2000	N/A	No	Yes
OSF-1 AD[ZRB+93]	Intel Paragon	OS	Yes	No
PScheD[LG97]	Cray T3E	User	Yes	N/A
SCore-D[HTI+96,HTI97b]	Workstation Cluster	User	Yes	No
SHARE[FPR96]	IBM SP-2	User	No	No

In this paper, *parallel process* is defined as a set of UNIX processes that are execution entities of a parallel program. A parallel process is a unit of gang scheduling. The frequency of communication in a parallel process can be much higher than that of distributed processes. At the same time, communication interface hardware is getting faster every year causing problems with system call overheads. To tackle this problem, there are several *user-level communication* proposals which allow users to access communication interfaces directly[PLC95,vEBV95,THIS97,CMC97].

User-level communication provides high-performance communication, however, it introduces a new problem when implementing gang scheduling. First, the network interface status must be saved and restored when switching processes. Second, some messages that should be received by a process before being switched may be received by another process after being switched. There should be some mechanism to avoid this situation.

We proposed *network preemption* to tackle the problem when implementing a gang scheduler, SCore-D[HTI+96,HTI97b], with a user-level communication library, PM[THIS97]. Network preemption can utilize gang scheduling without sacrificing user-level communication performance. PM is designed not only for providing high-performance communication, but also provides the required functions for network preemption. Our gang-scheduling scheme is all software; no special hardware support is assumed. SCore-D is designed for workstation and PC clusters, and is implemented as a set of daemon processes running on top of the UNIX operating system. SCore-D explicitly controls (schedules) user processes via UNIX signals. Thus, no kernel modification is required at all. With the network preemption, network status is saved and restored when switching parallel processes. However, the implemented gang scheduling overhead was evaluated with some simple programs, and was not analyzed[HTI+96,HTI97b].

In this paper, we evaluate the SCore-D gang scheduling overhead with more realistic applications, NAS parallel benchmark programs[BBLS93]. NAS parallel benchmark is a set of numerical programs, each of them is a component of CFD calculation. Thus they are expected to exhibit some aspects of real world

problems. The gang scheduling overhead is analyzed, and we found that the time of saving and restoring network context occupies more than two third of gang scheduling time on the applications with 64 processors.

2 Related Work

CM-5 has a hardware support for network preemption called *All-Fall-Down*[Thi92]. In the All-Fall-Down mode, all messages in the network fall down to the nearest node regardless of destination. To restore the network context, the fallen messages are reinjected into the network. Since the CM-5 network was not designed to preserve message order, the disturbance of message order by All-Fall-Down does not cause a problem. When the All-Fall-Down takes place, message order is not preserved, and message sending by the user program may fail since the message sending operation is not an atomic operation. The user program must handle these situations and extra communication overhead is introduced.

SHARE is a gang scheduler on IBM SP-2 [FPR96]. SHARE saves and restores network hardware context, however, it has no message flushing mechanism. Each message has a tag to identify the process receiving the message. If a message is delivered to the wrong process, then message sending fails. Since failure recovery must be handled by software, it introduces additional communication overhead.

We propose *network preemption* to tackle the problem when implementing a gang scheduler with user-level communication. Network preemption can provide high-performance communication to its user and can utilize gang scheduling without sacrificing communication performance. The proposed network preemption for user-level communication not only enables gang scheduling, but also provides a method to tackle some distributed process problems, such as; distributed termination detection, consistent checkpointing, and global garbage collection[HTI97b].

3 Cluster Software System

We have been developing a cluster software system for workstation and PC clusters. Figure 1 shows the software structure of our cluster software system. SCore-D is a gang scheduler on top of the UNIX operating system. PM is a low-level, high-performance communication library. In our cluster system, PM plays an important role for both providing high-performance communication to its user and implementing gang scheduling. One unique feature of SCore-D is that it is written in MPC++[Ish96], a multi-threaded C++. The distributed control structure objects are linked with MPC++ global pointers. The MPI communication library is also implemented[OHT+97].

3.1 SCore-D

Figure 2 shows the process structure of SCore-D and user processes. Here, *parallel process* is defined as a set of processes invoked from a single parallel program.

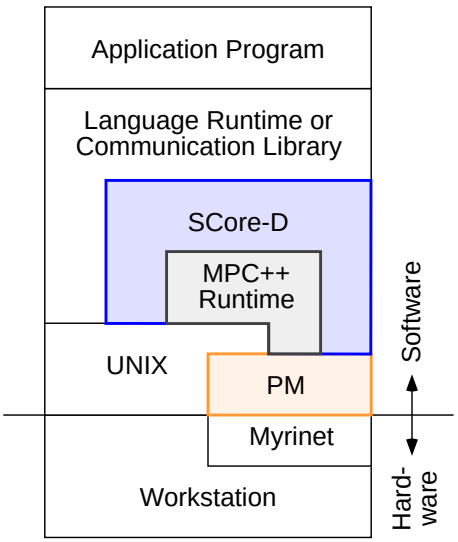

Fig. 1. SCore-D Software Structure

Each process of an SCore-D parallel process is running as a daemon process on every processor in a workstation cluster. Users can invoke their parallel program from their workstations. This invoked process on user's workstation is called the *Front End Process* (*FEP*). The FEP is a client for the SCore-D parallel computation server. Each process of a user parallel process is `fork`ed and `exec`ed by SCore-D processes. SCore-D can control user processes via UNIX signals.

Figure 3 shows an example of the SCore-D control structure to manage a user parallel process. Each user parallel process has a control structure tree. This control structure is distributed over a cluster to avoid a bottleneck. The root of the tree is a parallel process object, and the leaves are element process objects. Each element process object represents a UNIX process in a user parallel process. When a gang scheduler decides to stop a parallel process, then the stop command goes down the control tree, and finally every element process object sends a `SIGSTOP` signal to its corresponding process. The stopped state of each user process is caught by an element process object with a `wait()` system call, and the stopped event is forwarded to its super node. Each control node object synchronizes the events from its subnodes, and then forwards the event to its super node. When a parallel process object receives the events, it is guaranteed that all user processes are stopped. Along with the distributed control structure, commands are broadcasted and events are synchronized. Thus processes of a user parallel process change their states in a gang. The tree structure in Figure 3 is a binary tree. Actually, a hexadecimal-tree is used in SCore-D, because hexadecimal-tree is the fastest structure.

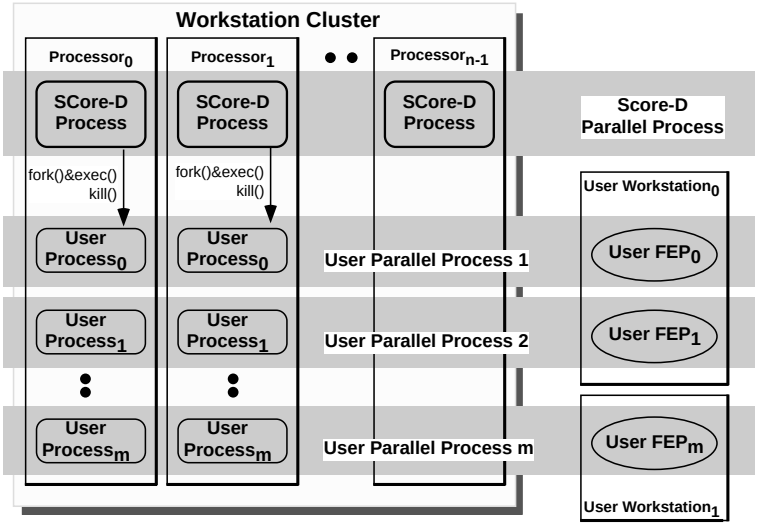

Fig. 2. SCore-D Process Structure

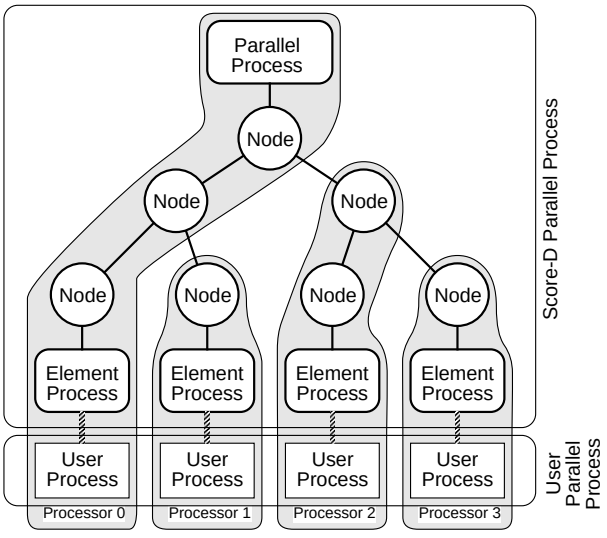

Fig. 3. Distributed Control Structure

One can implement any kind of scheduling policies and mechanisms with SCore-D. Actually, we have implemented a time sharing and space sharing scheduler, called DQT[HIK+95,HIN+95]. With DQT scheduling, sequential workload is treated as an exceptional case in that parallel process requires only one processor. However, all evaluations in this paper were done with simple time sharing scheduling for simplicity.

3.2 PM

PM is a low-level communication library for Myrinet[BCF+95]. PM[THIS97] consists of a software library, a software device driver, and firmware. It is designed to exploit the full potential of the Myrinet interface. PM is also designed so that a gang scheduler can be implemented on top of it. In this section, we will introduce two PM features that are designed for gang scheduling.

The first feature is multiple *channels*. A channel essentially consists of a pair of send and receive FIFO buffers. Those buffers are memory mapped to the address space of a process that opens the channel. This memory mapping technique reduces the number of memory copies and protects those buffers from access by other processes. While PM provides a connection-less communication model, the set of processors that can communicate with each other can be restricted when the channel is open. Inter-channel communication is not allowed. A channel is associated with a channel descriptor, and the descriptor can be passed to other processes like file descriptors in UNIX. SCore-D opens two channels; one for SCore-D itself and the other for user processes. The opened user channel descriptor is passed to the user process. PM also supports a blocking receive using receive interrupts. SCore-D waits for incoming message with blocking receive, and user processes wait for an incoming messages by polling. Thus the executions of SCore-D threads and a user process are interleaved at the SCore-D thread level.

The second feature of PM is its flow-control protocol. Myrinet supports hardware flow-control. However, relying on hardware flow-control can cause a deadlock because message sending region is locked until the end of the transmission, but the transmission is blocked by a hardware flow-control. PM's flow control protocol is called *Modified Ack/Nack*[THIS97]. PM provides an asynchronous message sending model. At the receiver processor, PM first determines if there is a enough room to hold the message in the receive buffer. If so, PM returns an *Ack* message. Otherwise PM returns a *Nack* message. On the sender side, when PM acknowledges with an Ack response, it frees the corresponding send buffer region. However, if a Nack message is received, PM resends the message and leaves the buffer region as is. Actually some Ack and Nack messages are merged into one to reduce message traffic. Refer to [THIS97] for more details. PM's channel and flow-control protocol introduces some important characteristics that must be considered when implementing gang scheduling.

Channel independence: When a receiver process falls into an infinite loop because of a program bug, it becomes so busy that it cannot handle the

messages in the receive buffer; therefore the sender process on the other processor eventually will be unable to send any message. On PM, however, communications through the other channels will take place normally. This is because PM's modified Ack/Nack protocol never stops the flow of messages. In this case, the Nack messages and resent messages are actually exchanged. This channel independence is very important when implementing a gang scheduler. To schedule processes in a gang, each scheduler process must be synchronized in some way. Having two network interfaces may provide two independent communication channels, but this requires extra investment in hardware. PM's multiple channel support and its channel independence avoid this.

Steady state of a channel: On a PM channel, if all Ack or Nack messages corresponding to all sent messages from a processor are received by the sender processor, then this means that there is no message being sent from the processor in the network. When a channel of a processor satisfies this condition, then the channel is referred to as being "in a *steady state*". When all processes associated with a parallel computation satisfy this condition, then there is no message associated with the computation in the network. Waiting for a steady state is different from waiting for transmission completion. The returning of a Nack message means that the message transmission has failed. When a receiver process is stopped, messages in the receive buffer are never consumed. Thus waiting for transmission completion may continue until the receiver process is resumed. When the transmission completion is applied to gang scheduling, a receiver process may be stopped by a signal, and the message flushing can last forever.

4 Gang Scheduling

Figure 4 shows the procedure for switching parallel processes in SCore-D. Parallel process switching consists of four phases.

Freeze Phase: Stopping user processes by sending `SIGSTOP`. SCore-D processes wait for stopped state of each user process with a `wait()` system call, and then wait until the user channels are in a steady state. Synchronizing the steady states of all the user channels guarantees that there are no messages from any user process in the network.

Save Phase: SCore-D saves the channel status of the user processes.

Restore Phase: SCore-D restores the channel status of the new user processes.

Run Phase: After being restored, SCore-D then sends `SIGCONT` signals to the new user processes. Eventually the user parallel job begins to run.

The entire set of channel contexts are called a *network context*, and the procedure described above is called *network preemption*, because SCore-D saves and restores network contexts. PM is designed to be preemptive, so that SCore-D can send signals to control user processes at any time. Thus user parallel processes can be preempted or killed by SCore-D at any time.

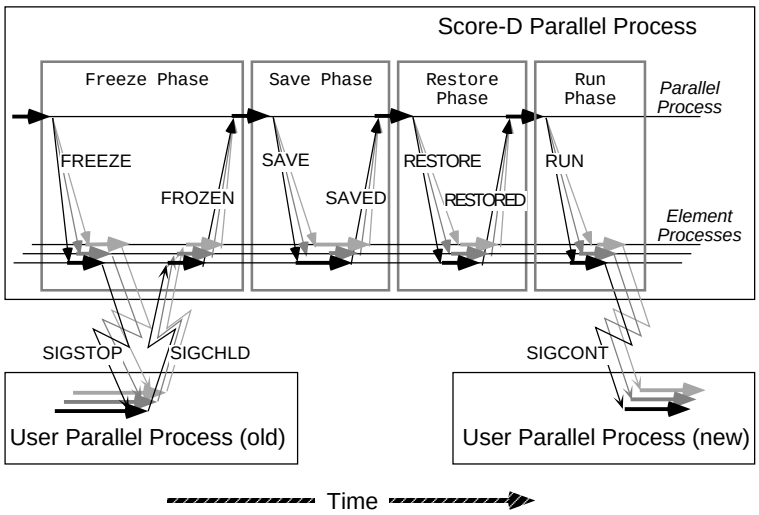

Fig. 4. Network Preemption

All broadcast and synchronization in the above four phases are propagated along with the distributed control structure in SCore-D. Thus, it takes order $log(N)$ time, where N is the number of processors, for each phase.

The network context is a snapshot of the network status that can be observed by software. Network preemption solves not only the problem of gang scheduling with user-level communication, but also helps for solving the following communicating distributed process problems.

Distributed termination detection: The detection of *no running process* in a set of distributed processes is well known as as the *distributed termination problem* and a number of algorithms have been proposed to tackle this (among these, [Mis83,CL85] are the most famous). The biggest difficulty of this problem comes from checking for the existence of messages in the network. With network preemption, this can be done by simply counting the number of messages in the saved network context. Distributed termination detection using network preemption has already been implemented in SCore-D-D[HTI97a]. SCore-D has also been successful in detecting deadlocked user processes.

Consistent checkpointing: When processes are gang-scheduled, the process contexts and channel contexts can be saved in permanent storage systems such as disks. Execution can be resumed by restoring those contexts. No additional mechanism is required to have a consistent state in the distributed processes.

Global garbage collection: Global garbage collection is difficult because of the references included in transmitted messages[KMY94]. As in the case of

consistent checkpointing, network preemption gives a clear consistent state timing and a chance to investigate the messages in transmission. Here, channel contexts are added to the root set of a global garbage collection. Thus network preemption makes the "marking of live objects" in a global garbage collection easy.

5 Evaluation

The overhead of the implemented gang scheduler was evaluated on our PC cluster II (Table 2). The PC cluster II consists of 64 PentiumPros (200MHz) connected by Myrinet (160MB/s bandwidth).

Table 2. RWC PC Cluster II

Number of Processors	64
Processor	PentiumPro
Clock [MHz]	200
Cache [KB]	512
Memory [MB]	256
I/O Bus	PCI
Network	Myrinet
Operating System	NetBSD 1.2
Min. Latency (PM) [μs]	7.5
Max. Bandwidth (PM) [MB/s]	117.6
Min. MPI Latency [μs]	12.0
Max. MPI Bandwidth [MB/s]	36.8

Table 3 shows the times to save and restore a PM channel context. In this table, "receive buffer full" means that the receive buffer holds 511 messages (65,408 bytes in total), and "send buffer full" means that the send buffer holds 255 messages (61,200 bytes in total). The saving time is larger than the restoring time, because read operations from PCI memory region takes longer than write operations.

The network context switching time depends on the number and the amount of messages in the network and in the receive and send buffers when gang scheduling takes place. Having larger receive and send buffers contributes to communication performance, however, it also increases the required network context switching time. The relation of buffer sizes to network context switching time is similar to the case of process context switching. The larger the register file size, the slower the process context switching.

NAS parallel benchmark programs (version 2.3, MPI)[BBLS93] are used for the evaluation of SCore-D gang scheduling. We selected EP, FT and CG from the benchmark (class A). EP is an embarrassingly parallel program; there is almost no communication. FT is a 3-D FFT program and is a communication bound

Table 3. Channel Context Save/Restore Time

Recv. Buffer	Send Buffer	Save [$msec$]	Restore [$msec$]
Empty	Empty	0.62	0.18
Empty	Full	1.96	1.56
Full	Empty	2.16	1.59
Full	Full	3.70	3.19

program. CG is a conjugate gradient method program; there are large amount of communication, but not so much as in the FT program. In short, these three programs exhibit different communication patterns.

In the current SCore-D implementation, SCore-D does not check if the next process to be scheduled is the same as the currently running process, and SCore-D naively switches one parallel process. Thus submitting one program is enough to measure gang scheduling overhead. Through the evaluation, there is no running process, but SCore-D and the process submitted via SCore-D.

Figure 5 shows the times of freeze, save and restore phases on the applications, measurement is for processors 8, 16, 32 and 64, with a time slice of $100\,msec$. The freeze phase time depends on the channel context sizes. And the times of all three phases depend on the number of processors, because each phase contains a broadcast and a barrier synchronization. In the EP program, there is almost no communication. Thus most of the processing time of each phase can be thought of as a base overhead of a broadcast and a barrier synchronization along with the distributed tree control structure.

With FT and CG programs, save and restore phase times are larger than that of EP programs. This comes from the size of channel contexts and is due to the larger amounts of communications. Since FT program is a communication bound program, the total gang scheduling times (sum of save, restore and freeze times) are the highest in all cases. Thus gang scheduling time depends on communication pattern of an application program running under SCore-D.

As described in Section 4, SCore-D gang scheduling consists of process context and network context switches. In the SCore-D implementation, these two context switching are mutually dependent and are not divisible. The time necessary for switching process contexts is reasonable (approximately $40\,\mu sec$) when compared with the time needed for switching network contexts. The time of saving and restoring network context occupies more than one third of gang scheduling time on applications in most cases.

The gang scheduling overhead observed by an application O is defined as

$$O = (T_{Gang} - T_{NoGang})/T_{NoGang}$$

Here, T_{Gang} is the execution time under the SCore-D gang-scheduler, and T_{NoGang} is the execution time with an infinite time slice. However, the calculated overhead is subject to measurement error if the differences in execution times are

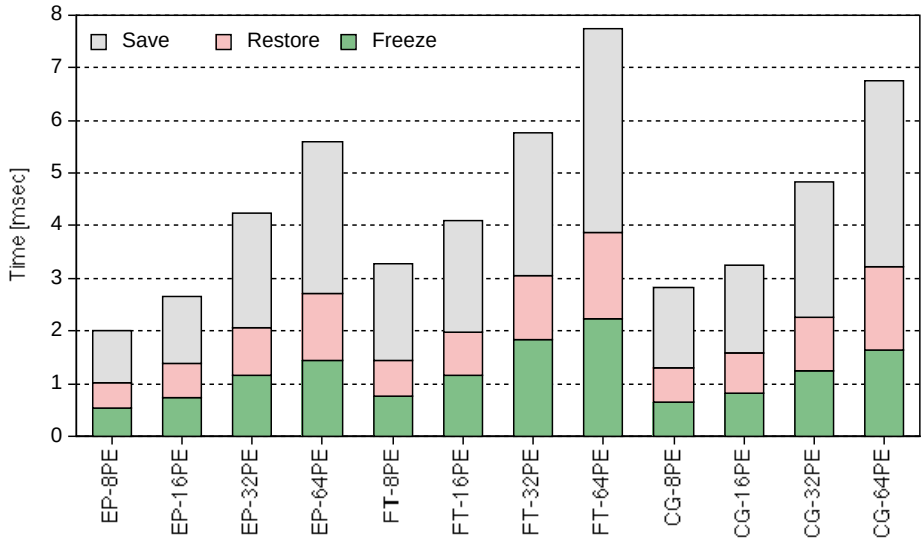

Fig. 5. Breakdown of Network Preemption

small. So we evaluated the gang scheduling overhead with smaller time slices (from 50 to 200 *msec*) to obtain larger elapsed time differences.

Figure 6 shows the gang scheduling overhead observed at each application. Here, the time slice of gang scheduling is set to 50, 100 and 200 *msec*, and the number of processors are 8, 16, 32 and 64. When the time slice is doubled, the slowdown is approximately halved.

The slowdown of program execution due to gang scheduling comes from a variety of reasons: SCore-D scheduling overhead, the cache effect, UNIX operating system overhead, co-scheduling skew[ADV+94], etc. Also scheduling overhead depends on the communication pattern of a user program.

Slowdown observed at the application level and the overhead measured at the SCore-D level can be different. One reason for this is that SCore-D can only detect the status changes of a user process with a `wait()` system call. There can be delay between signal sending and the detection of status change of process by the signal. Also co-scheduling skew can not be observed by SCore-D.

Comparing Figure 5 and Figure 6, we find that the overhead (slowdown) of the CG program is the highest, while the gang scheduling time of FT programs is the longest. We are now investigating these points.

6 Concluding Remarks

U-Net[vEBV95] and AM-II[CMC97] support *endpoints* similar to PM's channel to provide multiplexed, virtualized networks. If there were a sufficient number of channels (endpoints), there would be no need of network preemption when

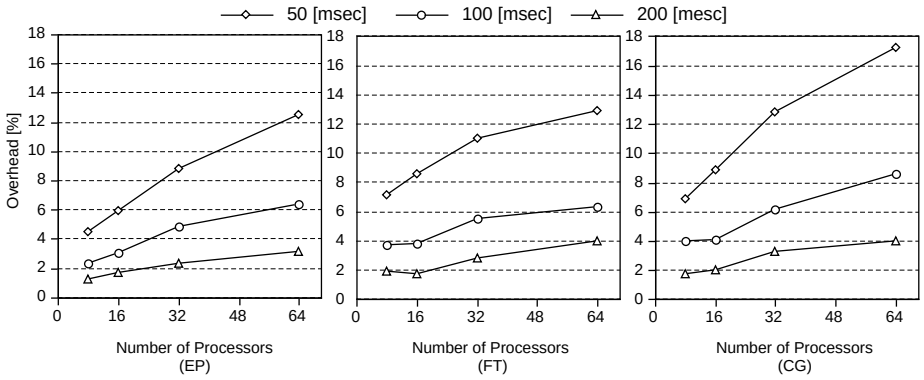

Fig. 6. Slowdown due to gang scheduling

gang-scheduling. However, one must guarantee that there is no message in all those channels involved in the parallel process in the network before the channel is reused. This means that the number of channels should be larger than the number of generated processes in the lifetime of the scheduling system. Therefore a message flushing mechanism is still needed.

Having a large number of channels degrades communication performance. User-level communication libraries have to check channels ready to send. The larger the number of channels, the larger the sending overhead. In AM-II, this problem is avoided by introducing endpoint scheduling[CMC97]. In contrast, PM limits the number of channels to four and provides channel context switching facilities. Simple round-robin scheduling for polling send buffers is used in PM. As described in Section 3.2, network preemption using channel context switching can be applied to a variety of problems in parallel processing.

Through the evaluation of our implementation on a PC cluster II, we confirmed that slowdown is less than 9 %with a 100 $msec$time slice. With a compute bound application, such as EP, the overhead is less than 7 %. We also confirmed that the overhead can be further reduced by increasing the time slice.

We found that much of the gang scheduling overhead comes from saving and restoring network context. Basically, saving and restoring network context is just copying memory between message buffers and the network context save area. It is expected, therefore, that gang scheduling overhead can be reduced by using a computer having a higher memory-copy performance.

It is normally assumed that gang scheduling overhead is quite high, and consequently time slice is longer than that of UNIX (Table 1). Although the overhead incurred by SCore-D gang scheduling is not small, we believe that it is acceptable. We have already confirmed that users can run interactive parallel programs on SCore-D[HTI97b].

Currently SCore-D is running on SunOS, NetBSD and LINUX. Our cluster software system including SCore-D, MPC++, PM, and MPI with PM is available at `http://www.rwcp.or.jp/lab/pdslab/dist/`.

References

[ADV+94] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. UC Berkeley Technical Report CS-94-838, Computer Science Division, University of California, Berkeley, 1994.

[BBLS93] D. H. Bailey, J. T. Barton, T. A. Lasinski, and H. D. Simon. The NAS Parallel Benchmarks. NASA Technical Memorandum 103863, NASA Ames Research Center, 1993.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[CL85] Mani Chandy and Leslie Lamport. Distributed snapshot: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CMC97] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnect'97*, August 1997.

[FPR96] Hubertus Franke, Pratap Pattnaik, and Larry Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Frontier'96*, pages 1–9, October 1996.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[GTU91] A. Gupta, A. Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *ACM SIGMETRICS*, pages 120–132, 1991.

[GW95] Brent Gorda and Rich Wolski. Time Sharing Massively Parallel Machines. In *1995 International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.

[HIK+95] Atsushi Hori, Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. A Scalable Time-Sharing Scheduling for Partitionable, Distributed Memory Parallel Machines. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. II*, pages 173–182. IEEE Computer Society Press, January 1995.

[HIN+95] Atsushi Hori, Yutaka Ishikawa, Jörg Nolte, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo. Time Space Sharing Scheduling: A Simulation Analysis. In S. Haridi, K. Ali, and P. Magnusson, editors, *Euro-Par'95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 623–634. Springer-Verlag, August 1995.

[HTI+96] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, Noriyuki Soda, Hiroki Konaka, and Munenori Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In D. G. Feitelson and L. Rudolph, editors, *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 76–83. Springer-Verlag, April 1996.

[HTI97a] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Global State Detection using Network Preemption. In D. G. Feitelson and L. Rudolph, editors, *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 262–276. Springer-Verlag, April 1997.

[HTI97b] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. User-level Parallel Operating System for Clustered Commodity Computers. In *Proceedings of Cluster Computing Conference '97*, March 1997.

[Ish96] Yutaka Ishikawa. Multi Thread Template Library – MPC++ Version 2.0 Level 0 Document –. Technical Report TR–96012, RWC, September 1996.

[KMY94] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient Parallel Global Garbage Collection on Massively Parallel Computers. In *Supercomputing Conference*, pages 79–88, 1994.

[LG97] Richard N. Lagerstrom and Stephan K. Gipp. PScheD Political Scheduling on the CRAY T3E. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 117–138. Springer-Verlag, April 1997.

[Mis83] J. Misra. Detecting termination of distributed computations using markers. In *Second ACM Symposium on Principles Distributed Computing*, pages 290–294, August 1983.

[OHT+97] Francis O'Carroll, Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, and Mitsuhisa Sato. Performance of MPI on Workstation/PC Clusters using Myrinet. In *Proceedings of Cluster Computing Conference '97*, March 1997.

[OSS80] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–105, February 1980.

[Ous82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[PLC95] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinoi Fast Messages (FM) for Myrinet. In *Supercomputing'95*, December 1995.

[Thi92] Thinking Machines Corporation. *NI Systems Programming*, October 1992. Version 7.1.

[THIS97] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In Peter Sloot Bob Hertzberger, editor, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer-Verlag, April 1997.

[vEBV95] Thorston von Eicken, Anindya Basu, and Werner Vogels. U-Net: A User Level Network Interface for Parallel and Distributed Computing. In *Fifteenth ACM Sumposium on Operating Systems Principles*, pages 40–53, 1995.

[ZRB+93] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John Lo Verso, Michael Leibensperger, Michael Branett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *San Diego Conference Proceedings of 1993 Winter USENIX*, pages 449–468, January 1993.