Implementation of Gang-Scheduling on Workstation Cluster

Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa,
Noriyuki Soda†, Hiroki Konaka, Munenori Maeda

Tsukuba Research Center
Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, JAPAN
TEL:+81-298-53-1661, FAX:+81-298-53-1652

† Software Research Associates, Inc.

E-mail:{hori,tezuka,ishikawa,soda,konaka,m-maeda}@trc.rwcp.or.jp
URL:http://www.rwcp.or.jp/people/mpslab/score/scored/scored.html

Abstract

The goal of this paper is to determine how efficiently we can implement an adequate parallel programming environment on a workstation cluster without modifying the existing operating system. We have implemented a runtime environment for parallel programs and gang-scheduling on a workstation cluster. In this paper, we report the techniques used to implement gang-scheduling on a workstation cluster and the problems we faced. The most important technique is "network preemption" and a unique feature of our approach is that the gang-scheduling is also written in a parallel language. Our evaluation shows that gang-scheduling on workstation clusters can be practical.

1 Introduction

Workstation clusters are gathering attentions to an alternative of parallel machines [1, 2, 13]. If a workstation cluster can be made to imitate a parallel machine, then it would be a cost-effective and familiar-to-use parallel execution environment. To prove this, we have implemented a parallel program execution environment on a workstation cluster.

The goal of this paper is to determine how efficiently we can implement an adequate parallel programming environment on a workstation cluster without modifying the existing operating system. We have implemented an efficient runtime environment for parallel programs and gang-scheduling on a workstation cluster.

Gang-scheduling is known to be efficient for job scheduling parallel programs [10, 4, 2]. However, it is not obvious how efficient it will be when implemented on workstation cluster.

In this paper, we report on techniques to implement gang-scheduling on a workstation cluster and problems we faced. The most important technique is "network preemption" and a unique feature of our approach is that the gang-scheduling is also written in a parallel language.

2 Assumptions and Terminology

Parallel Process
A "parallel process" is a set of processes that are execution entities of an SPMD program. When parallel processes are switched, all parallel processes are assumed to be scheduled simultaneously (gang-scheduled).

Workstation Cluster
A workstation cluster is a set of workstations connected by a high-speed network. Here, the workstation cluster is a computation server for parallel (and sequential) programs. If a user requests n processors to run a parallel program, the system provides at least n processors out of N processors in the cluster ($n \leq N$). We assume that every workstation in the cluster is

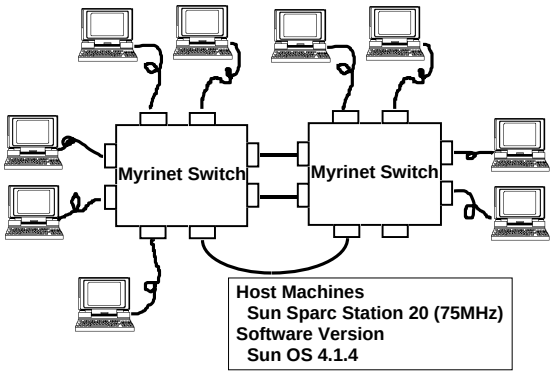

Figure 1: Myrinet and SS20 Cluster

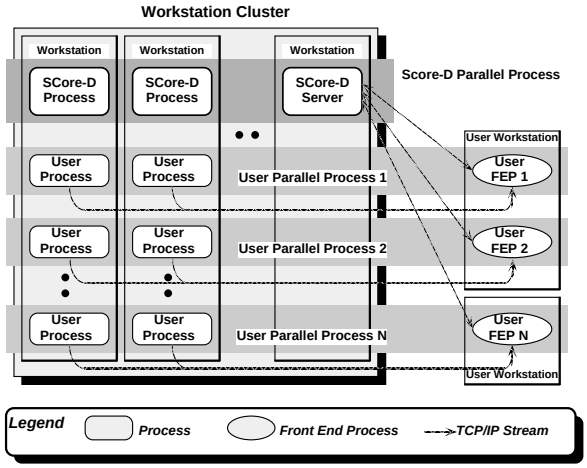

Figure 2: Process Structure of SCore-D

dedicated for use as a computation server. And we assume that none of the workstations are available for use as a personal computer.

This usage model is almost the same as the *Processor Pool Model*[14], except that our model can provide a "virtual parallel machine." The workstation cluster can be multiplexed in processor space and/or time to serve as a virtual parallel machine.

The current configuration of our workstation cluster is shown in Figure 1. Nine SS20s are connected by a Myrinet [3], a gigabit class high speed LAN. All the evaluations in this paper are measured on this workstation cluster.

Processor

In this paper, we assume all workstations have exactly one processor. This assumption simplifies the model used for explanation, and this is true of the workstation cluster we used.

3 SCore-D

SCore-D is a parallel process. Figure 2 shows the process structure on a workstation cluster with SCore-D. One of the SCore-D daemon processes is a dedicated server that is the entry point (connection host) from user programs. This process is called the "Server Process." The rest of the processes are called "SCore-D Processes."

Here, it is assumed that user programs are linked with an appropriate runtime library. When the user program is invoked on the user's local workstation, the runtime tries to make a TCP connection with the SCore-D server. The process running on the user's workstation is called the "Front End Process (FEP)." When the connection is accepted, the runtime passes information on the user's program to the SCore-D server, and the server passes the information to the rest of the SCore-D processes. Then SCore-D spawns (`fork` and `exec`, in UNIX) the user processes, and the user processes now become a parallel process. When spawned, each SCore-D process make a TCP connection with the FEP. This TCP stream becomes the standard output of the user parallel process, and the FEP passes the streamed data to its standard output. Due to the limitation of the current version, one of the workstations in the cluster is a dedicated SCore-D server.

The user parallel process can be stopped, run, and killed, when the SCore-D processes send, `SIGSTOP`, `SIGCONT`, and `SIGKILL` signals respectively to the child processes. SCore-D processes can also detect an abnormal end in user process. Using the signal functions of UNIX, SCore-D can schedule user parallel processes.

From the user's view point, it looks as if the invoked user program spawns itself onto the workstations in the cluster. Further, when the FEP is suspended, resumed or killed by the user, the corresponding user parallel process is suspended, resumed or killed. Thus the FEP and spawned parallel process look seamless.

3.1 MPC++

The SCore-D program is written in MPC++ . The user parallel programs running under SCore-D are assumed to be written in MPC++ . Since SCore-D itself

is a parallel program, it is natural to write in a multi-threaded parallel programming language.

MPC++ is a multithreaded parallel language based on C++[7, 9]. MPC++ provides a SPMD parallel programming model and threads of control to extract the full power of parallel machines or workstation clusters. One of the most unique features of MPC++ is meta-level programming which enables users to extend its language features [6, 8].

The MPC++ compiler consists of a front-end processor and a back-end processor. The front-end can also generate C++ source code and the generated C++ code can be compiled with `GNU g++` compiler. The MPC++ language features and the meta-programming results are thus compiled and will run on most computers.

3.2 MPC++ runtime library

The evaluated programs described in this paper are compiled with the MPC++ front-end processor and `GNU g++` compiler. The compiled codes are linked with MPC++ runtime library. The library is designed to have low overhead and to support the functions required by MPC++ programs.

There are two kinds of runtime library for MPC++ program. One is stand-alone and the other is assumed to run under SCore-D. The stand-alone version forks processes using the UNIX `rsh` command. We also implemented two ways to wait for incoming messages from remote processors, block-wait and busy-wait. The MPC++ process is blocked when there is no current message using block-wait. The MPC++ process continues spinning around until a remote message(s) arrive(s) by busy-wait. SCore-D uses the block-wait runtime library, while user parallel processes use busy-wait runtime library. To do so, the overhead to block user process can be avoided, and user processes can get more CPU time.

When a thread suspends and waits at synchronization point, its execution stack should be kept in the thread model of MPC++ . This is a normal thread model and `setjmp` and `longjmp` subroutines are used to implement thread context switching. It is known that `longjmp` takes longer time on the processor with register windows than the other kind of processors. We have succeeded to fasten `longjmp` several times faster, writing our own `setjmp` and `longjmp` subroutine in an assembler for SPARCstations.

The other key technique used in the MPC++ runtime library is our own Myrinet driver software, described in the next subsection. With the Myrinet software driver and the fast thread implementation, the MPC++ runtime library realizes a very efficient multi-threaded programing environment.

3.3 PM Communication library

The Myrinet LAN interface board has a dedicated processor, called LANai. LANai software is also provided by Myricom, Inc. Their focus is, however, on achieving high bandwidth. The latency from the user level Myrinet LAN is only a few times faster than UDP with 10 base-T. This is not surprising as they are targeting users using Myrinet as an alternative to Ethernet.

FM[11] achieved 22 μ sec in one-way latency with the same Myrinet, using their own LANai program. We have also developed our own LANai program and driver program, called PM. PM achieved 24 μ sec in one-way latency. Both FM and PM implement the communication layer at user-level, and use neither system calls nor interrupts. In the order of micro-seconds, the overheads incurred by system calls or interrupts are prohibitive.

PM can support a multi-process environment, while FM cannot. PM has several communication channels [1]. A channel consists of two FIFO buffers for receiving and sending messages. Since these channels are memory-mapped, copying messages is avoided. These channels can be used to implement priority in message sending and receiving, or to realize a multi-process environment. In SCore-D, SCore-D processes use one channel, and the user processes use the other channel(s). If those channels are memory-mapped in each user address space, then the inter-process protection can be guaranteed.

For flow-control, an `Ack` message is sent back to the sender when the receiving is succeeded. If the receive buffer is full, PM sends back a `Nack` message. At the sender, when the `Ack` message is received, PM just frees the sending message area of memory. Thus this flow control mechanism does not result in doubling the latency.

Due to the multi-channel support and the flow control, the PM performance is slightly degraded compared with FM. However, these are the keys to developing a multi-process environment, as described in the next subsection.

3.4 Support for gang scheduling

To support a multi-process environment, it is not enough to have multiple channels. Even if there were

[1] Currently three channels are implemented in PM.

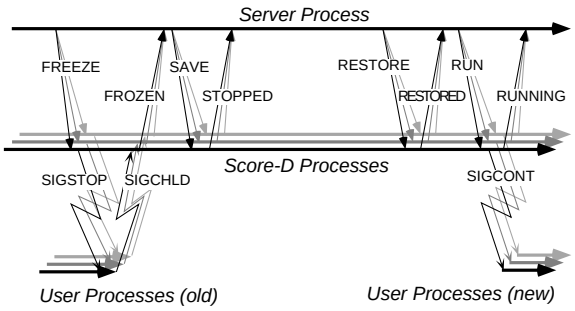

Figure 3: Parallel Process Switching

a sufficient number of channels, and the number of channels limits the number of simultaneously running processes, one should guarantee that there is no message in all those channels involved in the process in the network, when the channel is reused. Otherwise a message being sent in a parallel process may be lost and another parallel process may receive the message instead. This results in confusion.

If a message protocol layer exists in the operating system kernel, then this situation can be avoided. However, if all the message protocol layers are implemented at user level, it is very important to have fast inter-processor communication, and therefore we need a mechanism to detect when the messages for a parallel process in a network are flushed. If this is possible, then parallel processes can be preemptable, time-sharing of parallel processes can be implemented, and fast user-level communication is achieved at the same time.

To implement this requirement, we apply the `Ack` based protocol of PM to guarantee the non-existence of a message for a process in the network. It is necessary to sense the state in which PM (and LANai) receives all the `Ack` or `Nack` messages corresponding to the sending messages. We call this the "steady state." Further, PM provides the other functions to save and restore the channel context.

However, we must still guarantee that no message comes while or after the channel context is being saved. Unless loss of message or an inconsistent channel status would result. Figure 3 shows the procedure to switch processes in gang-scheduling which guarantee this. This procedure exactly mirrors the SCore-D process. We assume that the SCore-D server process has the control of process scheduling.

1. The server process decides to switch processes, and tells all SCore-D processes to stop currently running user processes (FREEZE message in Figure 3).
2. Each SCore-D process sends `SIGSTOP!$` and knows that the user process has stopped when it receives the `SIGCHLD` signal. Then SCore-D processes wait until the user's PM channel is in the steady state.
3. The server process is informed each processor is now in the steady state (FROZEN message). The server process waits until it receives a FROZEN message from each of the processors involved in user processes. This finally guarantees that there are no messages from user processes in the network.
4. After this guarantee, the server process tells all SCore-D processes to save the channel context (SAVE message).
5. Each SCore-D process saves its channel context. Completion of saving is reported to the server process (STOPPED message).
6. The server process waits for all STOPPED messages, to confirm that all user processes have been stopped. Then server process tells user processes to restore the context of the next process to run with the RESTORE message.
7. Each SCore-D process restores the channel context, and reports to the server process when done (RESTORED message).
8. When the server process has received all RESTORED messages, the server tells all SCore-D processes to run the new user processes (RUN message).
9. The SCore-D process sends a `SIGCONT` signal, and reports to the server process (RUNNING message).
10. The server process now knows that all new user processes are running.

The above procedure can be thought of as "network preemption." We have already proposed that this network preemption can be used not only for gang-scheduling, but also to detect an idle or terminated status in a parallel process, checkpointing, or global GC[5].

```
 1 int pe;
 2 int dist() {
 3   return( pe = ( pe + 1 ) % NPE );
 4 }
 5 int fibonacci( int n ) {
 6   if( n < 2 ) return( n );
 7   else
 8     return( fibonacci( n-1 )@[ dist() ] +
 9             fibonacci( n-2 )@[ dist() ] );
10 }
11 void fib( int n, int loop ) {
12   int i;
13   for( i=0; i<loop; i++ )
14     fib( n );
15 }
```

Figure 4: Example of MPC++ program

```
1 void rt( int hop_count ) {
2   if( hop_count == 0 ) exit( 0 );
3   rt( hop_count - 1 )@()[ next_pe() ];
4 }
```

Figure 5: Round-Trip program

4 Evaluation

Figures 4 and 5 are the MPC++ programs used to evaluate the SCore-D scheduling. Figure 4 is a fibonacci program to calculate the nth number in the fibonacci series. In this program, two threads are forked in a thread recursively (lines 8 and 9). The @ symbol at the end of the function call and the expression in square brackets indicates synchronous remote function call of the function on the processor specified by the expression.

The first thread to calculate the $n-1$th term in the fibonacci series is forked and waits for its answer. Then the second thread to calculate the $n-2$th term is forked and waits again. Finally the answers return to their parent thread. Those threads are distributed simply in a round-robin fashion. The `fib()` function is the top level function to be used in the evaluation. It simply iterates to calculate the fibonacci term for the number of times specified in the `loop` argument.

Figure 5 is the other MPC++ program used in the evaluation. In this program, threads are forked to the next processor sequentially. A thread forks another thread to the next processor and terminates. In the notation of thread invocation in this program, the function is forked asynchronously way (line 3). For more details, refer [6, 8, 9].

We chose those two evaluation programs because they are positioned on opposite sides in the execution pattern. The fibonacci program forks a number of threads almost explosively. In the round-trip program, however, there is no more than one running thread and no more than one message during the execution of the program.

Table 1: Execution time of evaluation programs [$sec.$]

	Number of processors			
	1	2	4	8
fib(15,1000)	15.47	77.7	105.1	121.5
rt(1000000)	4.09	34.55	35.56	34.94

Table 2: Network preemption time [$10^{-3} sec.$]

Buffer		Time	
Receive	Send	Save	Restore
empty	empty	0.13	0.11
empty	full	1.88	1.40
full	empty	3.39	1.95
full	full	5.15	3.22

4.1 MPC++ runtime performance

Table 1 shows the execution time for each evaluation program on our workstation cluster. For the fibonacci program, the larger the number of processors, the longer the execution time. This is because the granularity of the thread is too fine. In this paper, however, the communication and thread invocation pattern is the focus, not the speed. Supposedly, the execution time of a round-trip program is almost constant, independent of the number of processors.

From the execution time of the round-trip program we can estimate the overhead for MPC++ runtime. It takes about 4 μsec. to fork a local thread. For a remote thread, it takes about 35 μsec. including 24 μ sec. one way latency at the PM level. With the fibonacci program, we found that our MPC++ runtime is about 17 times faster than implementing the thread using LWP provided with the SunOS.

Table 2 shows the time to save or restore the network context at the PM level. In this table, buffer "empty" means that there is no message in the buffer, and "full" means that the buffer is almost full. A full receive buffer contains 2,730 messages in 32 KBytes. A full send buffer contains 511 messages in 12 KBytes.

As expected, the time to save or restore depends on the amount of messages in the buffers. In this table, context saving takes more time. This is because reading from the S-Bus memory space is slower than the writing to the S-Bus memory space.

Table 3: Elapsed Time Ratio under SCore-D

SCore-D/ stand-alone	Number of processors 1	2	4	8
fib(15,1000)	1.00	0.99	1.01	1.01
rt(1000000)	1.00	1.01	0.97	1.00

Time Quantum : Infinite

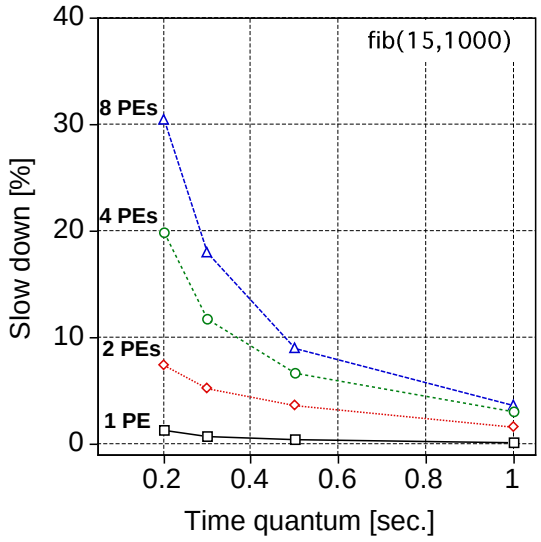

Figure 6: Gang-scheduling Overhead : fib()

Table 3 shows a comparison of elapsed time between the program linked with stand-alone runtime and the program running under SCore-D linked with SCore-D runtime. In this table, the time quantum of gang-scheduling is infinite. Since there is no reason for a slow-down in program execution under SCore-D, the speeds of the evaluation programs are the same.

4.2 Gang-scheduling performance

Figures 6 and 7 show the slow-down curves due to the gang-scheduling overhead on each evaluation program. The time quantum is varied between 200, 300, 500, and 1,000 msec. In each time quantum, the number of processors is also varied between 1, 2, 4 and 8.

The slow-down due to the scheduling overhead can be calculated as,

$$T_{Elapsed} = \frac{T_{Quantum}}{T_{Quantum} - T_{Overhead}} T_{Process}$$

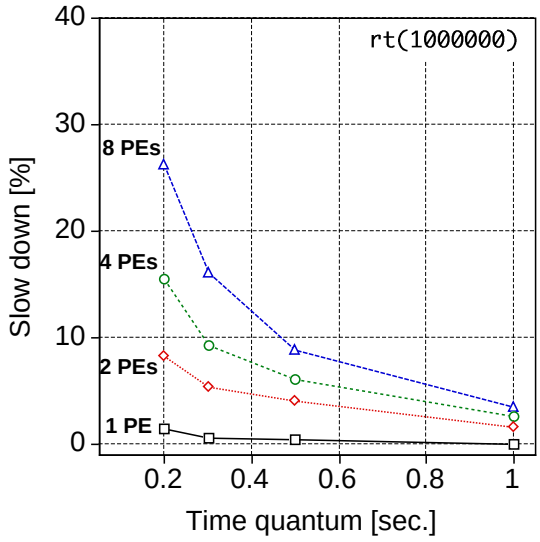

Figure 7: Gang-scheduling Overhead : rt()

Table 4: Gang-scheduling Overhead [$10^{-3}sec.$]

Program	TQ	Number of processors 1	2	4	8
fib(15,1000)	200	2.6	13.9	33.1	46.6
	300	2.3	14.7	31.5	45.5
	500	2.3	14.7	31.5	45.5
	1000	1.6	16.1	29.9	35.1
rt(1000000)	200	3.0	15.3	27.0	41.6
	300	1.9	15.2	25.6	41.5
	500	2.5	19.5	28.8	41.0
	1000	0.2	15.8	25.8	34.5

where $T_{Elapsed}$ is the elapsed time, $T_{Quantum}$ is the time quantum, $T_{Overhead}$ is the scheduling overhead, and $T_{Process}$ is the processing time. Figure 8 shows the slow-down curves calculated with this formula, and the scheduling overhead times calculated from the evaluation results are shown in Table 4.

The possible reasons for scheduling overhead are, i) SCore-D overhead, ii) refilling the cache (flushed out by SCore-D), iii) saving and restoring the network context (network preemption), and iv) process switching at the UNIX level. The SCore-D overhead includes the costs of broadcasts and synchronizations. Thus the scheduling overhead can depends on the number of processors involved. The cost of network preemption depends on the number of messages and the total

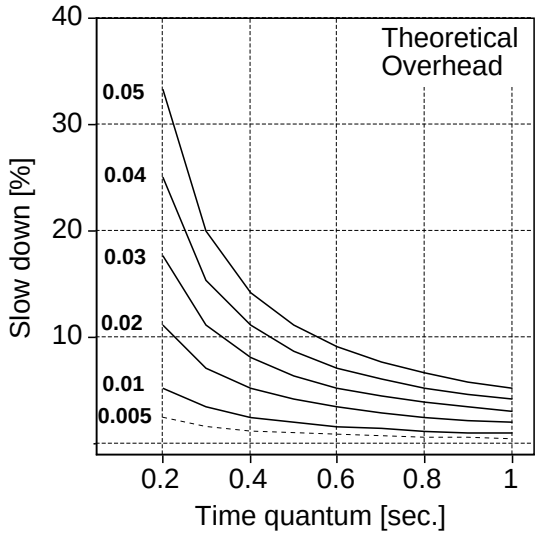

Figure 8: Gang-scheduling Overhead [$10^{-3}sec.$]

message size, as shown in Table 2. As with the round-trip program and any program with one processor, the overhead from network preemption can be neglected.

As shown in Figures 6 and 7, and also in Table 4, the scheduling overhead depends on the number of processors. According to our investigation, most of the overhead comes from the delay of `SIGSTOP` signal. Curiously the `SIGSTOP` signal to suspend other processes is delayed, and the delay time varies up to 50 msec. This phenomenon can not be found in the IRIX System V.4 (SiliconGraphics) or FreeBSD 2.0.0. We suppose that the SunOS (version 4.1.4) intentionally delays delivery of `SIGSTOP` signal until the end of the time quantum. Since the server process should wait for all process to stop, the time to stop a running parallel process is depending on the number of processors involved, and costly. Furthermore, the variance in `SIGSTOP` delivery creates "coscheduling skew"[2]. Both of these can be severe problems when implementing gang-scheduling on a workstation cluster.

4.3 Voluntary gang scheduling

To avoid the delay in signal delivery, we implemented another version of gang-scheduling. The runtime library of user processes yields by itself when the time quantum ends. We call this version "voluntary gang-scheduling." Table 5 shows the overhead calculated in the same way as for Table 4. The overhead is reduced roughly three times or more at 8 processors.

Table 5: Voluntary Gang-scheduling Overhead [$10^{-3}sec.$]

Program	TQ	Number of processors 1	2	4	8
fib(15,1000)	200	3.4	6.4	13.4	15.3
	300	2.7	4.5	13.2	14.3
	500	2.7	6.7	11.0	11.7
	1000	3.7	5.0	13.3	11.0
rt(1000000)	200	2.8	5.2	11.6	11.9
	300	2.3	5.6	11.7	11.2
	500	2.3	7.3	8.7	10.5
	1000	4.1	8.6	12.2	8.8

And the dependence on the overhead with the number of processors is weakened. When the time quantum is one second, the slow-down due to the scheduling overhead is less than 1.4 %.

5 Concluding Remarks

SCore-D and the MPC++ runtime contribute to an efficient parallel program execution environment. The gang-scheduling of SCore-D realizes multi-user, multi-parallel-process environment. To implement efficient and practical gang-scheduling, we developed "network preemption."

It can be very difficult to estimate and guarantee the maximum time in a large network, considering the effect of hot-spots [12]. This situation becomes a severe problem in implementing real-time scheduling. We have already proposed an architectural support for gang-scheduling, called "Drain"[5]. The Drain mechanism can guarantee the maximum time to reach the steady state.

We found that the signal delivery of the SunOS can be an obstacle when implementing gang-scheduling. However it can be avoided with the voluntary gang-scheduling. With network preemption and the voluntary gang-scheduling, we believe that gang-scheduling on a workstation cluster can be made sufficiently practical and scalable.

The target of SCore-D is very similar to that of GLUnix [1, 2]. In [2], some simulated results of gang-scheduling on a workstation cluster are shown. However, this paper is the first report on implementing gang-scheduling on a worksation cluster as far as we know.

The other unique feature of SCore-D is that SCore-

D itself is written in MPC++ , a multi-threaded programming language. All the functions described in this paper have been implemented in only 1,600 lines of code.

SCore-D will support global resource management including parallel I/O. We intend to move on larger workstation cluster, and we will continue to investigate the implementation of gang-scheduling.

References

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, et al. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] 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 Berekeley Technical Report CS-94-838, Computer Science Division, University of California, Berekeley, 1994.

[3] 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.

[4] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *COMPUTER*, 23(5):65–77, May 1990.

[5] Atsushi Hori, Takashi Yokota, Yutaka Ishikawa, Shuichi Sakai, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jörg Nolte, Hiroshi Matsuoka, Kazuaki Okamoto, and Hideo Hirono. Time Space Sharing Scheduling and Architectural Support. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 92–105. Springer-Verlag, April 1995.

[6] Yutaka Ishikawa. MPC++: Massively Parallel, Message Passing, Meta-Level Programming C++. In *Parallel Object Oriented Methods and Application'94*, 1994.

[7] Yutaka Ishikawa. The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1. Technical Report TR–94014, RWC, June 1994.

[8] Yutaka Ishikawa. Meta-Level Architecture for Extendable C++. Technical Report TR–94024, RWC, January 1995.

[9] Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, and Jörg Nolte. MPC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

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

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

[12] Gregory F. Pfister and V. Alan Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, pages 943–948, October 1985.

[13] Jim Pruyne and Miron Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, April 1995.

[14] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.