Global State Detection using Network Preemption

Atsushi Hori Hiroshi Tezuka Yutaka Ishikawa

Tsukuba Research Center
Real World Computing Partnership
1-6-1 Takezono, Tsukuba-shi, Ibaraki 305, JAPAN
TEL:+81-298-53-1661, FAX:+81-298-53-1652
E-mail:{hori,tezuka,ishikawa}@trc.rwcp.or.jp
URL:http://www.rwcp.or.jp/

Abstract

Gang scheduling provides shorter response time and enables interactive parallel programming. To utilize processor resources on interactive parallel programs, global state of distributed parallel processes should be detected. This problem is well-known as "distributed termination problem." In this paper, we propose a practical method to detect a global state of distributed processes. There are two key methods to detect a global state described in this paper. One is by network preemption and the other is by combining global state detection with gang scheduling. The overhead for the detection of a global state is negligible. To implement our scheme, we extend our gang scheduler, SCore-D, to an operating system by implementing a system-call mechanism. We confirmed that our proposed global state detection scheme is very useful on SCore-D.

1 Introduction

Gang scheduling is thought to be effective especially for fine-grain parallel programs [16, 5]. The most benefit of gang-scheduling is that it can provide shorter response time than batch scheduling. Taking hours of running time can be reduced to minutes on a parallel machine or a workstation cluster. Thus, gang scheduling enables some sort of parallel applications to be interactive [6].

In UNIX, a state transition of a process is clearly defined. An idle state of a process means that there exists a blocked system-call. However, the state transition of gang-scheduled processes can be different from that of sequential processes. There could be a case in that processes are gang-scheduled every time a system-call is blocked. This strategy introduces a number of gang context switches proportional to the number of processors. Considering the overhead of gang context switch, this situation should be avoided. So it is desirable that the processes are gang-switched when every process is idle or blocked.

The detection of "no running process" in a set of communicating processes is well known as as a "distributed termination problem." A number of algorithms have been proposed to tackle this distributed termination detection problem [4, 15]. Essentially, there are two key points to detect the global termination of a distributed parallel processes. One is to detect the termination of all processes. The other is to guarantee that there exists no message for the processes in the communication network. If these predicates are true, then the processes are assumed globally terminated. Some algorithms require extra *marker* messages to detect the non-existence of messages, some require counting the number of messages, and so on. Those additional mechanisms add extra overhead to the underlying computation.

Globally terminated or idling distributed processes should be detected by a parallel operating system. The operating system is responsible for not allocating resources in vain. These idling processes can be preempted and switched to another runnable distributed processes. Of course, a globally terminated process should be terminated by the operating system too.

We have developed a user-level parallel operating system, called SCore-D [10, 9], for workstation clusters. The key idea enabling gang scheduling in SCore-D is *network preemption*. Under SCore-D, distributed user processes can communicate by manipulating network hardware directly. When switching processes, the messages are flushed out from the network. Then the flushed messages and the network hardware status are saved and restored on each processor.

We applied this network preemption technique to the global state detection problem. Each time processes are gang-switched, the saved network status is investigated on each processor. When every process is idle and there is no message in saved network context, then the distributed processes are assumed to be globally terminated. The benefits of this method are that the overhead to detect a global state is negligible, and there is no need of a special mechanism for the underlying computation.

Here, we slightly modify the definition of the global termination by considering the viewpoint of parallel operating systems. In addition to the conditions of detecting a global termination, if there exists at least one blocked system-call, then the distributed process is assumed to be globally idle. After the system-call is done, then the distributed processes become ready to run. Most of the blocking system-calls causing global idle situations are I/O operations. For a multi-threaded system, it could be argued that the blocking time can be hidden by scheduling other threads. Disk related I/O operations usually end within after tens of milli-seconds, and there could be a sufficient quantity and/or length of threads in a distributed processes. However, there is also the case waiting for human reaction, quite a problem if he or she has been out at a restaurant having a lunch. It is hard to imagine that there can always be a sufficient quantity and/or length of threads to fill several hours.

We implemented a system-call mechanism and some I/O devices in the SCore-D. In addition to the global termination detection, we also implemented global idle detection. Eventually, we confirmed that this global idle detection is very effective for maximizing system throughput and for decreasing user frustration.

2 SCore-D Overview

At this time, the nature of parallel operating systems is still unclear, it is important to implement and evaluate the proposed parallel operating system functions. To catch up state-of-the-art hardware technology, fast development is crucial. Therefore development a parallel operating system at user-level is reasonable for parallel operating system research. SCore-D is a user-level parallel operating system on top of UNIX [10, 9].

Figure 1 shows the software structure of SCore-D. A unique feature of SCore-D is that it is written in

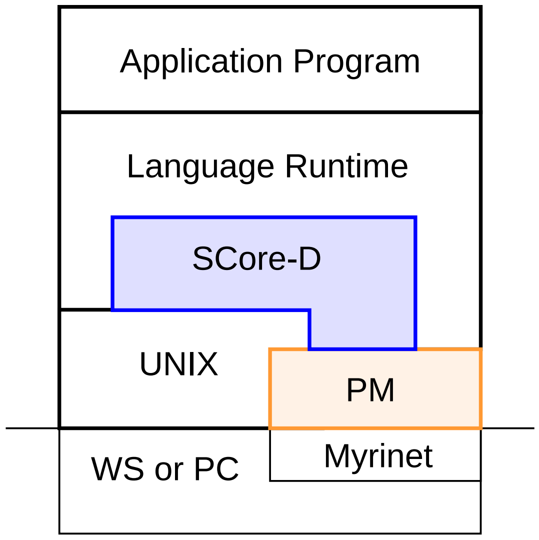

Figure 1: Software structure of SCore-D

MPC++ [11], a multi-threaded C++. Although the runtime library of MPC++ is hidden in this figure, The runtime system of MPC++ has been developed for SCore-D and user programs. The runtime system of the other parallel object-oriented language, OCore[14], is now supported by SCore-D.

Currently, SCore-D is running on both our workstation cluster (36 SparcStation 20s) and PC cluster [8] (32 Pentium PCs). Myrinet [3] is used as the interprocessor communication hardware in both clusters. We have developed Myrinet driver software, called PM [19]. PM supports not only low-level communication, but also some functions for network preemption. SCore-D is independent of network hardware and the programming language of application programs. We are now developing an Ethernet communication library too.

To execute a user program, a user has to connect to a SCore-D server via TCP on Ethernet and supply the required information to execute the user program, including user ID, the current directory path, the executable filename, number of processors, and so on. If the user program is linked with an appropriate runtime library, then all procedures take place in the runtime library. SCore-D then `forks` and `execs` user program over the processors. After user processes are forked, the runtime library initializes the PM library according to information passed from SCore-D, and the user program starts. For gang scheduling and job control, user processes are controlled via UNIX signals

Table 1: PM performance

Machine	Clock $[MHz]$	Latency $[\mu s]$	Bandwidth $[MB/s]$
SS20	75	8.0	38.6
Pentium	166	7.2	117.6

Table 2: PM interface

Send and Receive	
`GetSendBuf`	allocate send buffer
`Send`	send a message
`Receive`	check message arrival
`PutReceiveBuf`	free receive buffer
Network Preemption Support	
`SendStable`	confirm message arrivals
`SaveChannel`	save network context
`RestoreChannel`	restore network context
`GetChannelStatus`	return channel status

Table 3: Context Save and Restore Time

SS20	save	restore
Send Full	3747	3002
Recv Full	7203	5201
Both Empty	1866	1094
Both Full	8963	7073
Pentium	**save**	**restore**
Send Full	2399	1843
Recv Full	4663	2955
Both Empty	938	365
Both Full	6118	4427

$[\mu sec]$

by SCore-D [10].

2.1 PM

PM consists of a Myrinet firmware (LANai program) and a low-level communication library [19]. PM allows its users to access Myrinet hardware. Avoiding system-calls to UNIX and interrupts (whenever possible), we succeeded in reducing software overhead dramatically. The same technique can be found in [17]. Table 1 shows the one-way latency and bandwidth on SparcStation 20s (SS20) and PC-ATs. Although the Myrinet link has a bandwidth of 160 MB/s, PM's bandwidths are limited by the I/O bus speed used in these machines.

Table 2 shows a list of functions related to communication and network preemption. Communication with PM is asynchronous. To send a message, first a message buffer should be allocated by calling `GetSendBuf()`. Next the runtime constructs or packetizes a message in the allocated buffer area. Then the message buffer is sent to a destination processor by calling `Send()`. This allocation and sending procedure can avoid extra memory copying. The receiving procedure is almost the same as the sending. The lower half functions in Table 2 are for network preemption and are used in SCore-D only. PM supports multiple communication channels. The function `GetChannelStatus()` is to obtain channel information including the numbers of messages waiting for sending and receiving.

Table 3 shows the time required to save and restore network context. The time to save and restore depends on the number of messages and the total amount of message size in the receive and send buffers in a channel. In this table, "Send Full" means the send buffer contains 511 messages (49,056 bytes in total), and "Recv Full" means receive buffer contains 4,095 messages (65,520 bytes). Larger buffer size contributes communication performance, however, it takes longer time to save and restore network context.

SCore-D initializes Myrinet hardware and uses one channel for its inter-processor communications, while the user processes use the other channel. When SCore-D creates a new user processes, it resets the other channel for the user's inter-processor communication. With this multiple channel feature of PM, SCore-D and user process can share a Myrinet hardware. All the send and receive operations in PM are safe from preemption of CPU at any time, so that SCore-D can preempt user process at any time.

From the viewpoint of SCore-D, the functions in Table 2 is an API. SCore-D does not assume having Myrinet interface, but assumes a communication library having the same functions defined in Table 2.

2.2 Network preemption

Since PM allows its users to access network hardware directly, the network hardware status is also saved and restored when switching processes. However, this is not enough. There exists the possibility of receiving a message belonging to the process before switching. To avoid this, the messages in the network should be flushed out before starting a new process.

PM uses a modified Ack/Nack protocol for flow-control. Although message delivery is reliable on Myrinet, a flow-control mechanism is still needed. The allocated sending message region is preserved until the corresponding Ack message is received. Note that these Ack/Nack messages are used for releasing or re-sending the corresponding message. The `Send()` function sends a message asynchronously. Thus, latency and bandwidth can be improved. We applied this protocol to the detection of message flushing. The `SendStable()` function returns if there exists a message in transient.

When user processes are gang-scheduled, SCore-D first sends UNIX signal `SIGSTOP` to all user processes. On each processor, SCore-D waits until the user process stops, then calls the `SendStable()` function to wait for flushing of messages sending from the node. The completion of the flushing at each node is synchronized in a barrier fashion, and the flushing of user messages in a network is then complete. The next thing to do is to save the network context on each processor, and to restore the network context of a new process. After reloading a new network context, the execution of new processes are resumed by sending `SIGCONT` signals.

CM-5 has a hardware support for network preemption, called *All-Fall-Down* [20]. When an operating system decides to switch processes, it sets a special hardware register to trigger the All-Fall-Down. In All-Fall-Down mode, all messages in the network fall down to the nearest processor 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. PM preserves message order. Message order is preserved even when network preemption takes place. PM also guarantees reliable communication. These message order preserving and reliable communication eliminate avoids the needed software overhead to handling irregular situations.

Franke, et al. also implemented a gang-scheduler, called *SHARE* on IBM SP-2 [7]. They also save and restore network hardware context. The communication mechanism used in [7] does not guarantee reliable communication. The absence of reliability may not require the network preemption. When a context switch takes place and a message arrives at a wrong process, then the message is discarded and then its sender must resend the message. The PM guarantees reliable communication, and this feature not only simplifies the programming, but also contributes the low-latency and high-bandwidth in communication shown in Table 1. The PM's reliable and asynchronous features require extra mechanisms, the network preemption including global synchronization, for enabling gang scheduling. However, considering that the frequency of communication is mush higher than that of gang scheduling, lower-overhead communication is desirable.

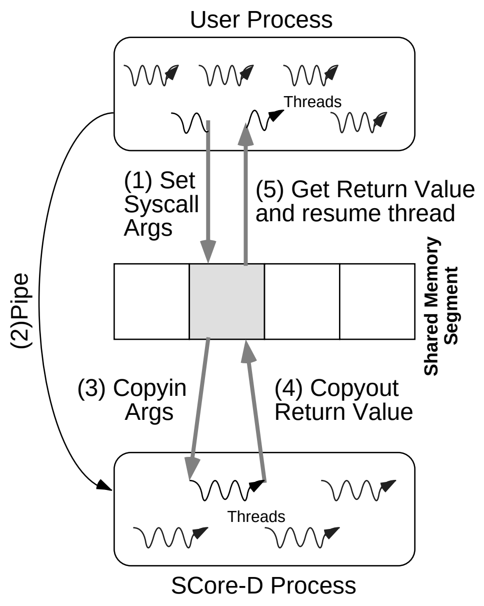

Figure 2: Systemcall mechanism

2.3 System-call mechanism

In [10], SCore-D is introduced as a gang scheduler. In addition, we have added the system-call mechanism and have implemented I/O mechanisms and other operating system functions. In this subsection, we briefly introduce the system-call mechanism implemented in SCore-D.

SCore-D and a user process share a memory segment defined in System V IPC. This memory segment is used to relay information to initialize a user process. The segment is also used for system-calls from user processes to SCore-D (Figure 2). The arguments for a system-call are copied into a region of a shared segment. SCore-D is notified of the existence of a service request via a UNIX pipe. SCore-D processes the request, and the result is copied to the same region.

```
void idle_loop( void )
{
  char *recv_buf;

  while( 1 ) {
    if( !Receive( channel, &recv_buf ) )
    {
      idle_flag = FALSE;
      process_message( recv_buf );
      PutReceiveBuf( channel );
      break;
    }
    if( syscall_cell->status == DONE )
    {
      idle_flag = FALSE;
      syscall_count --;
      enqueue_thread(
        syscall_cell->thread );
      break;
    }
    idle_flag = TRUE;
  }
  return;
}
```

Figure 3: Skeleton of an idle loop

2.4 Thread model

SCore-D assumes that user programs are written in a multi-threaded language and that the thread is implemented at the user-level. Figure 3 is a skeleton of idle loop for such a thread runtime library. We put an integer variable `idle_flag` in the shared memory segment to indicate if the user process is idle or not, so that SCore-D can observe its status.

Suspending a thread invoking a system-call, the thread runtime system can obtain the benefits both of kernel threads and user threads. SCore-D assumes that user programs run with a user-level thread runtime library. There is no need of a UNIX system-call to switch threads, and the runtime only suspends the thread invoking a system-call. In an idle loop, shown in Figure 3, a flag indicating the end of a system-call is checked. If a system-call is finished, then the runtime re-schedules the suspended thread. No thread is preempted when a system-call is done. Thus, our thread model can avoid the critical section problem discussed in [1].

3 Global State Detection

Although the asynchronous nature of PM contributes to communication performance, it complicates the detection of a global state. The overhead to detect a global state should be as low as possible. Some additional mechanism needed in user programs is not ideal. Here, we propose a practical method to detect a global state by combining network preemption for gang scheduling and global state detection.

In Figure 3, the `idle_flag` variable is located in the shared memory segment. The `syscall_count` variable counts pending system-calls and is also located in the shared segment. Now, the SCore-D can detect a global state in user's distributed processes each time the processes are gang-scheduled.

1. Suspend user distributed processes.
2. Preempt the network.
3. Gather message count information in preempted network context, the processes' activities, and the system-call count from each process.

As a result, if no message can be found in the preempted network, and there is no busy process, then the distributed processes are assumed to be either globally idle or terminated. Further, if there is no pending system-call, then the processes are assumed globally terminated, otherwise the processes are globally idle.

The first two steps in the above procedure follow the normal procedure for gang scheduling, and is nothing new in the detection of global state. Only the last step is modified. However, with normal gang scheduling, the last step is still needed to synchronize all network preemptions at each process. The overhead for the global state detection is only in adding extra arguments for the synchronizer. Thus the overhead for adding this global state detection mechanism is negligible.

The most drawback of this proposed method is that a global state can be detected only when distributed processes are gang-switched. This means that when a distributed processes becomes globally idle, the processors are idle for half of the time quantum in average. In many cases, users may expect a shorter response time with their interactive programs. To get a shorter response time, the process switching should take place in a shorter interval. However, the shorter the interval, the larger the overhead. To answer the question of how often, we have to know how much overhead incurred by the network preemption and the degree of user frustration concerning the interval. The measured overhead of gang scheduling under SCore-D is shown in Section 4.

Generality

So far, we have been targeting multi-threaded programs. We assume that the local idle state of user process can be detected by SCore-D. The `idle_flag` in Figure 3 should be set with great care. The `idle_flag` is set before incoming message is consumed by `PutReceiveBuf()`, and before the system-call post-processing. Otherwise, a user program can be mistaken as being globally idle or terminated.

Here, let us imagine a program in which a single thread per processor is running on a distributed shared memory machine, and the program code is busy-waiting at a global flag that may be set by another process. The `idle_flag` can be set in the busy-wait loop. However, a problem can arise over the race between the setting of the `idle_flag` and the consuming of a remote-memory-write message. At the moment just after the message to set the flag arrives and when the code is about to check the flag, a preemption may occur. If there is no message in the network and all the other processes are idle, then the program is mistaken as being in an idle state.

Thus, the generality of the proposed method of global state detection depends on the detection of a local state and timing of message consumption. At this moment, we are targeting multi-thread programming environments, and we can not go into further detail on shared memory programming models.

4 Gang Scheduling Overhead

The longer the time quantum, the lower the gang scheduling overhead. However, longer time quantums means a longer response time and an increased possibility of idling processors. Thus the time quantum of gang scheduling is a trade-off between these and should be decided in accordance with its overhead.

We have developed two types of clusters. One is a workstation cluster consisting of 36 SparcStation 20s (Figure 4). The other is PC cluster consisting of 32 PCs (Figure 5) [8]. Myrinet is used to interconnect the processors in each cluster.

Table 4 shows slowdown (overhead) due to the gang scheduling, varying the time quantum, 0.2, 0.5, and 1.0 second. To measure the overhead, we run a special program in which barrier synchronization is iterated 200,000 times running on 32 processors. Evaluating with this special program, all possible gang scheduling overhead, including co-scheduling skew[2], can be included. The slowdown is compared with the same program running with a stand-alone runtime library (called SCore-S) of MPC++ . On both our workstation and PC cluster, the slowdown is 8.84 % and 4.16 % with the time quantum of half second, respectively.

Figure 4: Workstation cluster

Figure 5: PC cluster

The overhead of PC cluster is much less than that of workstation cluster. We guess this difference is dominated by a scheduling policy of the local Unix operating system[10].

Although the time quantum is relatively larger than that of Unix, granularities of execution time of parallel applications considered to be larger than that of sequential applications. It is not reasonable to run a text editor on parallel machines. Scheduling tens of processor to echo one character is meaningless. Thus, even for an interactive parallel programming, processing granularities triggered by input commands should be larger, and the time quantum around one second is considered to be still acceptable. The mechanism of the global state detection of distributed processes can utilize processor resource and can reduce users frustration with interactive parallel programs.

Table 4: Slowdown due to gang scheduling [%]

	Time Quantum [*Sec.*]		
	1.0	0.5	0.2
Workstation Cluster	6.96	8.84	28.7
PC Cluster	2.87	4.16	6.25

The overhead of gang scheduling under SCore-D is not small. Sampling the global state of a user program for each gang scheduling is assumed to be practical.

5 Related Works

The global state detection problem can also arise in the area of distributed database [13], consistent checkpointing [18], and global garbage collection [12]. The most famous Chandy and Lamport algorithm to take a snapshot of a global state [4] requires $O(n^2)$ messages, where n is the number of processors, to check a global state. Although the number of required messages can be reduced when a network topology and router hardware knowledge are given [18], the detection mechanism should be triggered periodically. The other algorithm proposed by Misra proposed to use a *marker* message to guarantee non-existence of message in the network [15]. Passing one marker message at a time is required, and this marker message passing still adds extra overhead.

In most global detection algorithms proposed so far, it is assumed that messages in transient can not be observed. However, the proposed network preemption mechanism enables this with negligible extra overhead. As mentioned, global idle detection is especially effective for gang-scheduled interactive parallel programs. Thus, integrating a global state detection mechanism into a gang-scheduler is very natural. The message flushing mechanism implemented in PM is needed not only for the network preemption, but also for barrier synchronization. In many data parallel programming model, barrier synchronization is used to guarantee that 1) the procedure on every processor has reached at the synchronization point, and 2) remote memory write requests have been reflected to destination memory area. To guarantee the latter item, the PM's message flushing mechanism can be applied. In this case, the proper arrivals of all sending messages should be guaranteed. While in the case of network preemption, the non-existence of message including Ack/Nack messages should be guaranteed, no matter if sent messages are properly received.

As described, the most drawback of the proposed method is that the global state can only be detected at each time quantum. To investigate the global state, the network preemption is sufficient, but the saving and restoring network status are not needed. A global state can be detected by only stopping user processes and counting the flushed messages. More frequent investigation of global state than the gang switching can reduce the possible idle time. As shown in Table 3, it takes several milli-seconds to save and restore network context. This idea is feasible.

6 Concluding Remarks

We showed a practical method implemented in SCore-D, that of integrating global idle or termination detection with network preemption. With the proposed method, the overhead to detect a global state is negligible, although there is the drawback of being an off-line (or, sync-and-stop in terms of [18]) algorithm.

On shared memory model programs, our proposed method may not be applicable. However, global idle or termination detection is very important to utilize processor resources on interactive parallel programs. Thus, global state detection should be implemented in a parallel operating system.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management

of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. 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] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

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

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

[6] D. G. Feitelson and L. Rudolph. Parallel Job Scheduling: Issues and Approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, April 1995.

[7] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Frontier'96*, October 1996.

[8] A. Hori and H. Tezuka. Hardware Design and Implementation of PC Cluster. Technical Report TR–96017, RWC, December 1996.

[9] A. Hori, H. Tezuka, and Y. Ishikawa. Global State Detection using Network Preemption. In *Cluster Computing Conference '97*, March 1997.

[10] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. 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.

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

[12] T. Kamada, S. Matsuoka, and A. Yonezawa. Efficient Parallel Global Garbage Collection on Massively Parallel Computers. In *Supercomputing Conference*, pages 79–88, 1994.

[13] E. Knapp. Deadlock Detection in Distributed Database. *Computing Surveys*, 19(4):303–328, December 1987.

[14] H. Konaka, Y. Itoh, T. Tomokiyo, M. Maeda, Y. Ishikawa, and A. Hori. Adaptive Data Parallel Computation in the Parallel Object-Oriented Language *OCore*. In *Proc. of the International Conference Euro-Par'96, Vol.I*, pages 587–596, 1996.

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

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

[17] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinoi Fast Messages (FM) for Myrinet. In *Supercomputing'95*, December 1995.

[18] J. Plank. *EFFICIENT CHECKPOINTING ON MIMD ARCHITECTURES*. PhD thesis, Printceton University, 1993.

[19] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A High-Performance Communicatin Library for Multi-user Parallel Environments. Technical Report TR–96015, RWC, November 1996.

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