TR-660

A Scheme for State Change in a Distributed Environment Using Weighted Throw Counting

Kazuaki Rokusawa
rokusawa@okilab.oki.co.jp
Systems Laboratory, Oki Electric Industry Co., Ltd.
4-11-22, Shibaura, Minato-ku, Tokyo 108, JAPAN

Nobuyuki Ichiyoshi
ichiyoshi@icot.or.jp
Institute for New Generation Computer Technology
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

This paper proposes a scheme for changing the execution state of a pool of processes in a distributed environment where there may be processes in transit. The scheme can detect the completion of state change using weighted throw counting and detect the termination as well. It works whether the communication channels are synchronous or asynchronous, FIFO or non-FIFO. The message complexity of the scheme is typically O(number of processing elements).

1 Introduction

This paper proposes a scheme for changing the execution state of a pool of processes in a distributed environment. It works under the *weighted throw counting* (WTC) scheme [1] [1] for termination detection. The completion of state change can be detected while not losing the ability to detect termination.

We assume a distributed environment where multiple computational tasks are executing simultaneously. Each task is distributed over many processing elements, and each processing element can be assigned more than one task.

In such an environment, one sometimes needs to stop a task (to examine the computational state, or to collect statistics), restart it, change the priority of a task (to execute efficiently), or abort a task (when it goes into an infinite loop, etc.). We call such actions *state change* actions. They are different from general stability detection [5] which is superimposed on the basic computation and is not intended to bring changes in the basic computation.

[1] This scheme has been employed in a parallel implementation [7] of KL1 [8], a concurrent logic programming language, on the Multi-PSI [9].

Our scheme can detect the completion of state change and works whether the communication channels are synchronous or asynchronous, FIFO or non-FIFO. The state change scheme is implemented on top of the WTC mechanism. The capability of detecting termination (including termination during state change) is not lost.

This paper is organized as follows. Section 2 defines the computation model and the state change problem. The WTC scheme is described briefly in section 3. Section 4 presents a straightforward solution using distributed snapshots [6]. Our scheme using weighted throw counting is described in section 5. Finally, the comparison of our scheme with the straightforward one is given in section 6.

2 Computation Model

Our process control scheme is intended to work in a distributed computation environment where there may be multiple computational tasks running simultaneously. We model such an environment as follows:

- There are a finite number of process pools in the system (process pools represent independent computational tasks);
- A process pool consists of one controlling process and a finite number of child processes;
- Each process pool is assigned a unique process pool identifier (PID);
- Each child process has one of the n execution states: $S_1, S_2, \ldots, S_n$;
- A child process can terminate at any time;
- A child process can generate another child process having the same PID and the same state,

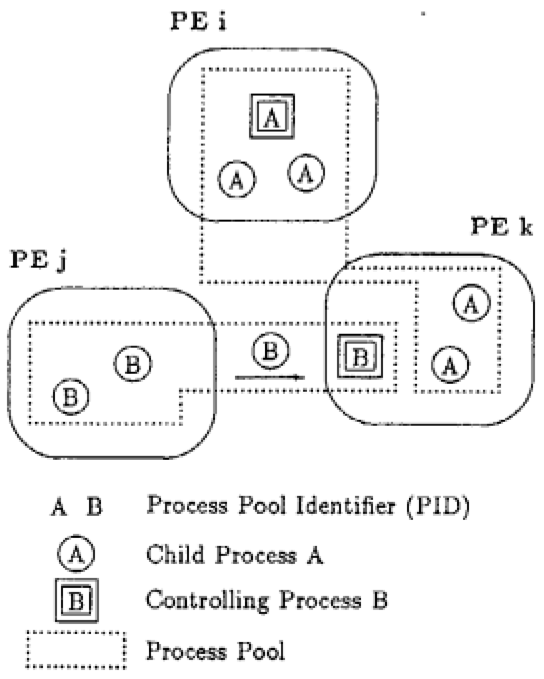

Figure 1: Computation Model

and a new process pool having a new PID as well. On the creation of a process pool, all child processes have the same state.

In this paper, "process" means "child process" unless otherwise indicated. A process pool *terminates* if all the children terminate.

A process pool described above is distributed over the following machine:

- A finite number of processing elements (PEs) interconnected by a communication network;
- No global storage; PEs may communicate by passing messages;
- Asynchronous communication, in which messages are delivered with arbitrary finite delay.

It is assumed that a PE can detect the termination of all processes in it having the same PID, and can change their states. The controlling process and PEs can communicate in both directions. A PE may send a message to the controlling process informing it of the termination of all processes, and the controlling process may send a message which forces processes to change their states.

Processes may migrate among PEs for load balancing; a PE may throw a process in the PE to another PE. As the thrown process is delivered with arbitrary finite delay, processes may be in transit at a given time (see figure 1).

Definition of the Problem

Initially, all processes in the same process pool are in the same state. *State change* is to change all processes in one process pool from one state to a new state, and is completed when the states of all the processes have changed into the new states. We require that two state changes not overlap, that is, a state change action not be initiated until the previous state change has completed.

To satisfy the above, a state change scheme must be able to:

- change states of all processes belonging to a particular process pool into a specified state, and
- detect the completion of a state change.

Note that the possible of existence of processes in transit (processes migrating from one PE to another) makes the completion detection nontrivial.

3 The WTC Scheme

This section gives a brief explanation of the weighted throw counting (WTC) scheme [1] [2], which is a distributed termination detection scheme. The scheme is an application of the weighted reference counting [3, 4], which is a garbage collection scheme for parallel processing systems, and can efficiently detect termination without probing or acknowledgement.

A non-empty set of processes in one PE having the same PID forms a subpool of processes, which is called a "process subpool", or a "subpool" in short. On receiving a thrown process, the PE decides whether there is already a subpool having the same PID as the thrown process. If there is, the PE adds the received process to the subpool; otherwise, it creates a new subpool.

We associate *weight* with the controlling process, each subpool and each process in transit. The weight of a subpool and that of a process in transit are positive integers, while the weight of the controlling process is a negative integer. The WTC scheme maintains the invariant that:

The sum of the weights is *zero*.

This ensures that the weight of the controlling process reaches zero if and only if all processes terminate, i.e., there is no processes neither in a PE nor in transit.

When all processes in it are terminated, the subpool terminates and sends a *terminated* message to

[2] Essentially the same scheme named *Credit Recovery* algorithm is presented in [2]. *Credit* in [2] corresponds to *weight* in the WTC scheme.

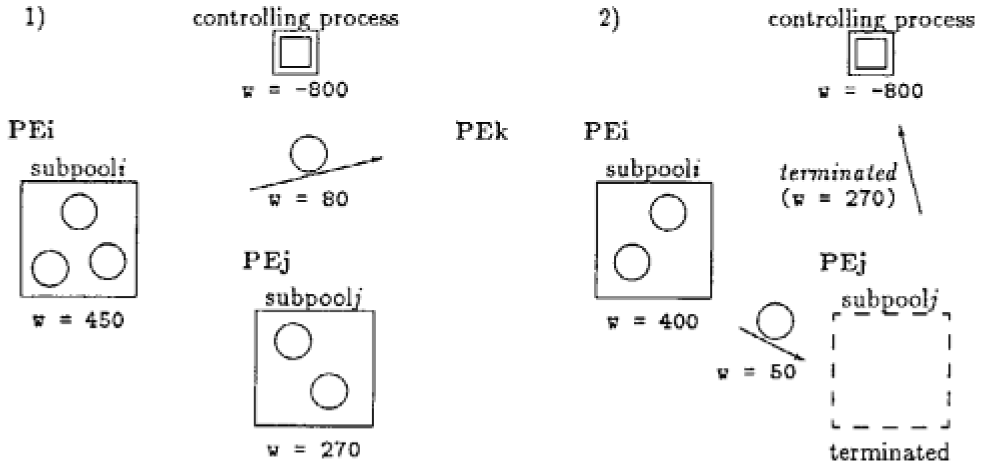

Figure 2: The WTC Scheme

the corresponding controlling process. This *terminated* message carries the weight of the terminated subpool. On receiving a *terminated* message, the controlling process adds the weight to its (negative) weight. If the weight of the controlling process reaches zero, the termination of all processes is detected (see figure 2).

4 Straightforward Solution

This section describes a straightforward scheme using distributed snapshots [6].

State change can be divided into the following two phases:

- Changing the state of all processes belonging to the pool;
- Detecting the completion of state change.

The former phase is performed by broadcasting and memorizing. The controlling process broadcasts a *change* message which carries the PID of the pool and a new state. When a PE receives a *change* message, it changes the state of processes belonging to the specified pool and keeps the new state with the PID. If there is no corresponding process, the PE only memorizes the PID and the new state. When all processes in a pool terminate, the PE still keeps the new state with the PID. On receiving a process with different state, the PE also changes the state of the received process into new one memorized. Therefore, a process with old state in a PE changes when a *change* message arrives, and a process with old state in transit changes when it reaches a PE with new state.

The latter phase is performed by the technique of *repeated observations* using distributed snapshots. After broadcasting a *change* message, the controlling process starts distributed snapshots, and repeats it until the completion is detected.

5 Solution Using the WTC scheme

As described in previous section, if the controlling process broadcasts a message and each PE memorizes a new state, it is ensured that state change completes in finite period. Therefore, only the detection of the completion of state change is required.

As mentioned in section 2, the operation of changing into the new state begins on the assumption that all processes have the old state.

5.1 Detection of the Completion

We show here how to detect the completion of state change using the WTC scheme.

Before state change, the controlling process allocates a variable called *changed weight* and sets its initial value at the (negative) weight of the controlling process. This *changed weight* plays an important role; it indicates the weight of processes with old states.

The controlling process broadcasts a *change* message. On receiving a *change* message, the PE changes the states of all processes in the corresponding subpool having the specified PID and sends back a *changed* message in acknowledgement. It carries the copy of the weight of the subpool, which indicates the sum of the copies of the weights of processes whose states have already been changed. If there is no corresponding subpool, the PE memorizes the PID and the state carried; which is equivalent to the creation of an empty subpool, a subpool with no weight.

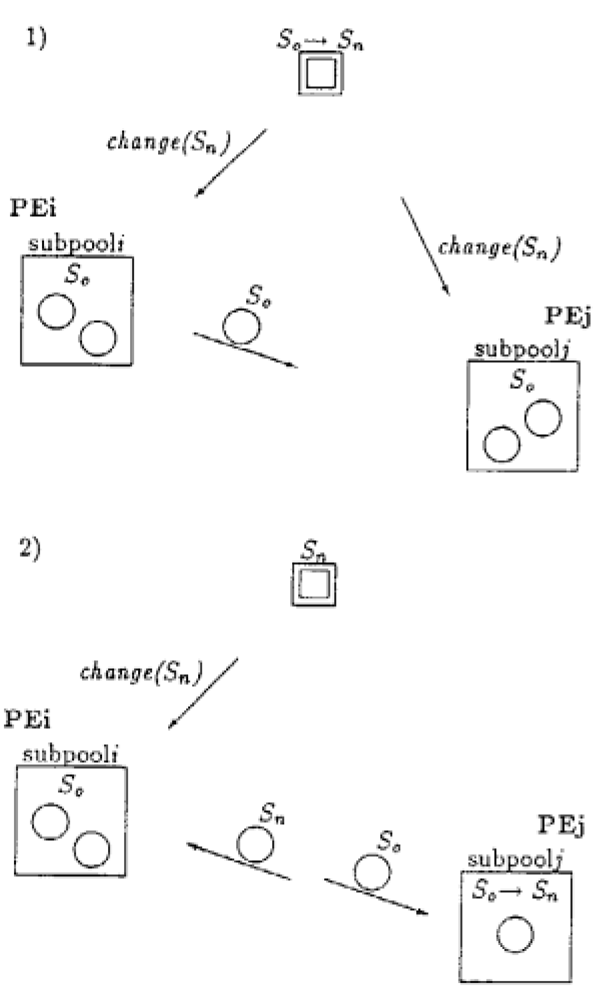

Figure 3: Which state is new ?

When a PE receives a thrown process having different state from the one of the corresponding subpool (including an empty subpool), it also changes the state of the received process and sends a *changed* message to the controlling process which carries the copy of the weight of the received process.

On receiving a *changed* message, the controlling process adds the received weight to the (negative) weight of the *changed weight*. Since the weight of the *changed weight* indicates the sum of the copies of the weights of processes whose states remain old, when the *changed weight* rearches zero, the completion of state change is guaranteed.

5.2 Situations to be Considered

Although the scheme mentioned above can detect completion of state change correctly, it is still incomplete because the following situations are not considered.

Which state is new ?

Since messages are delivered with arbitrary finite delay, a PE can receive a thrown process with not only old state but also either new state or same state. In figure 3, **PEj** (subpool*j*) receives a thrown process with old state (S_o), while **PEi** (subpool*i*) receives a thrown process with new state (S_n) if the thrown process reaches **PEi** earlier than the *change* message does.

When the state of a received process is old, the state of the process is changed into the (new) state of the subpool and the copy of the weight of the process should be sent back. On the other hand, when the state of a subpool is old, the state of the subpool should be changed and the copy of the weight of the subpool should be sent back.

Therefore, when a PE receives a thrown process with different state from the one of the corresponding subpool, it must decide which state is new. To perform this, we associate *generation* with each state. The controlling process holds a generation which is incremented on each state change and carried by a *change* message to PEs with a state. Each subpool holds a generation and assigns it to a thrown process. Since there are only two generations of state in a process pool during state change, three generation numbers are enough for generation control; for example, $0 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow \ldots$.

Termination during state change

Since a subpool can terminate at any time, the controlling process can receive two kinds of *terminated* messages; a *terminated* message sent before receiving a *change* message and sent after receiving a *change* message. In figure 4, the former and the latter are *terminated* messages sent by **PEi** and **PEj** respectively.

As the former carries the weight of a terminated subpool having old state, it is necessary to add the weight to the *changed weight*. This operation is similar to the action when receiving a *changed* message. On the other hand, as the latter carries the weight of a subpool with new state, the weight should not be added to the *changed weight*.

Thus, operations of the controlling process must differ in different *terminated* messages. Assigning generation to each *terminated* message makes it possible to the controlling process to discriminate between two kinds of *terminated* messages described above.

change message in transit

In case a *change* message reaches a PE having no corresponding subpool, no weight is sent back. Therefore, even if the *changed weight* reaches zero,

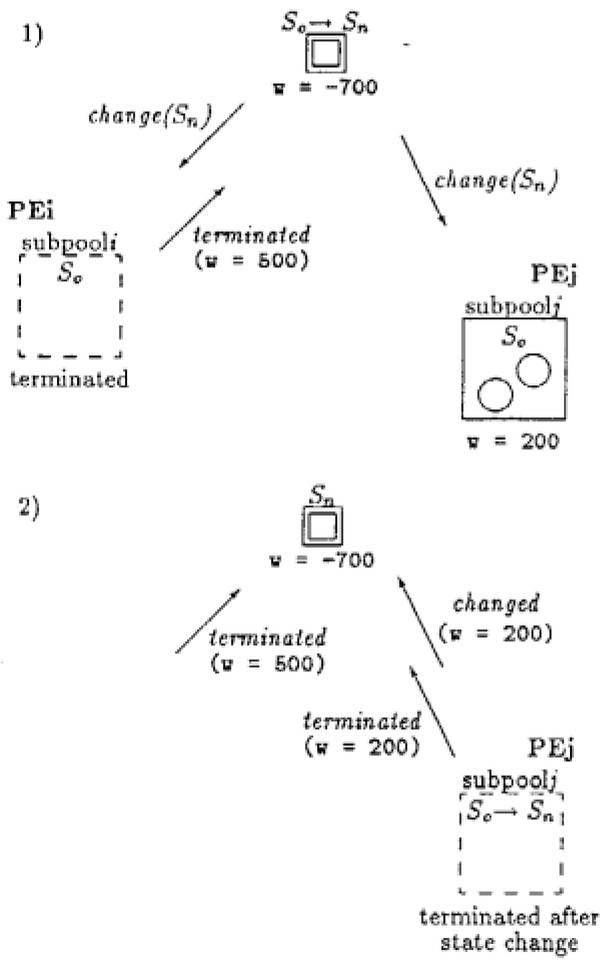

Figure 4: Two kinds of *terminated* messages

it is not sure that all *change* messages are received. Since messages are delivered with arbitrary finite delay, termination leaving change messages in transit is quite dangerous. If the PID is reused and new process pool with the PID is created, wrong state change may occur. To avoid the wrong termination detection, each PE sends back a *ackChange* message when it receives a *change* message.

5.3 The Whole Scheme

In consideration of situations mentioned above, the scheme is revised. The whole scheme is as follows (see figure 5).

Operations of the Controlling Process

On start of state change, the controlling process performs the following operations:

1. Allocates two variables called *changed weight* and *ack counter*, and sets initial values at the weight of controlling process and the number of the PEs respectively; The value of the *ack counter* indicates the number of *ackChange* messages which are not received;
2. Increments the generation;
3. Broadcasts a *change* message which carries the PID and new state with the incremented generation.

On receiving a *ackChange* message, the controlling process decrements the *ack counter*.

On receiving a *changed* message, the controlling process adds the received weight to the (negative) weight of the *changed weight*.

On receiving a *terminated* message with the *old* generation, the controlling process adds the received weight to both its weight and the *changed weight*.

On receiving a *terminated* message with the *same* generation, the controlling process adds the received weight to only its weight.

Detection of State Change: When both values of the *changed weight* and *ack counter* reach zero, the state change is completed; there is no processes with old state neither in a PE nor in transit, and no messages concerned (*change*, *changed* and *ackChange*) in transit.

Termination Detection: When the completion of state change is detected and the weight of the controlling process reaches zero, the controlling process broadcasts a *forget* message. On receiving a *forget* message, the PE forgets the PID and the state memorized, and sends back an *ackForget* message to the controlling process. When receiving all *ackForget* messages, the termination of the process pool completes.

Operations of Each PE

On receiving a *change* message, the PE sends back an *ackChange* message and performs one of the following operations:

- If the state carried by the *change* message is different from the one of the subpool with the specified PID, the PE changes the states of all processes in the subpool into the specified new state, and sends back a *changed* message which carries the copy of the weight of the subpool.
- If the state is the same as the one of the subpool, nothing is done. This is the case of having received a thrown process with new state before receiving the *change* message.

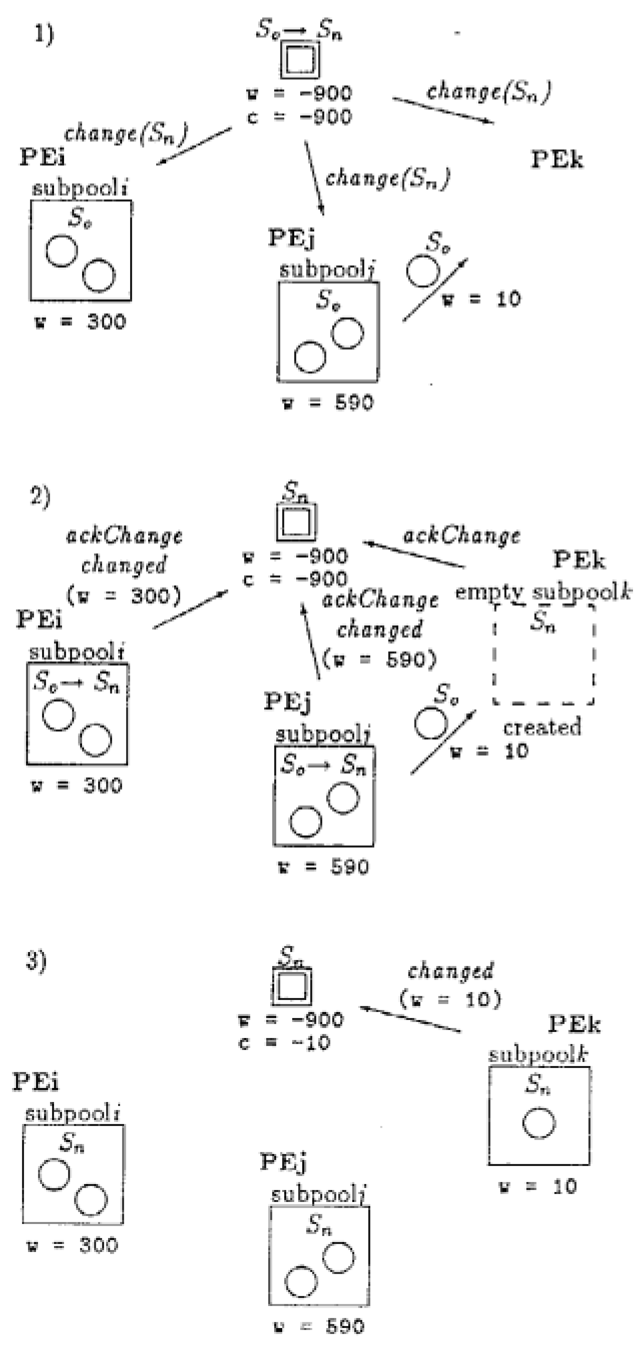

Figure 5: State Change Operations

- If there is no corresponding subpool, the PE memorizes the PID and the new state; which is equivalent to the creation of an empty subpool.

On receiving a thrown process with the *same* state, the PE only adds the weight of the process to the weight of the corresponding subpool. No message is sent.

On receiving a thrown process with the *different* state, after performing either of the following operations, the PE adds the weight.

- If the generation of the state of the subpool is *newer* than that of the process, the PE changes the state of the process and sends a *changed* message which carries the copy of the weight of the process.
- If the generation of the state of the subpool is *older*, the PE changes the state of the subpool and sends a *changed* message which carries the copy of the weight of the subpool.

When a subpool terminates, after sending a *terminated* message with the generation of the terminated subpool, the PE still keeps the PID and the state; which is equivalent to remain an empty subpool.

5.4 An Efficient Variant

Assigning a weight with a *change* message makes an *ackChange* message needless. An assigned weight is regarded as a weight of a thrown process with old state. An *ackChange* message can be merged into a *changed* message or replaced by a *terminated* message.

On start of state change, the controlling process performs the following operations:

1. Increments the generation;
2. Broadcasts a *change* message with a weight and subtracts the sum of the weights from the weight of the controlling process;
3. Allocates a variable called *changed weight* and sets initial value at the weight of the controlling process after subtraction.

When a PE receives a *change* message, it adds the weight of the *change* message to the weight of the subpool having the specified PID, and performs either of the following operations:

- If the state carried by the *change* message is different from the one of the subpool, the PE changes the states of all processes in the subpool into the specified new state, and sends back a *changed* message which carries the copy of the weight of the subpool.
- If the state is the same as the one of the subpool, the PE sends back a *changed* message which carries the copy of the weight of the *change* message.

If there is no corresponding subpool, the PE creates an empty subpool and sends back a *terminated* message with old generation which carries the weight of the *change* message.

6 Comparison

The straightforward scheme presented in section 4 has the following disadvantages:

- It only works under FIFO communication.
- Since all communication channels must be traced by a *marker* message [6], a large number of *marker* messages are needed. In the model defined in section 2 where all PEs can communicate each other, $n(n-1)$ *marker* messages are sent among n PEs. Thus, the message complexity becomes $O(n^2)$.

In contrast, our scheme has the following advantages:

- It works under both FIFO and non-FIFO communication.
- Both a *change* message and a *changed* message in response to a *change* message are sent n times. Although a *changed* message is also sent when a process with old state arrives, the number of processes in transit is in proportion to n in general. Therefore, the message complexity is $O(n)$.

7 Summary

We have devised a scheme for changing the execution state of a pool of processes in a distributed environment. Our scheme can detect the completion of state change using weighted throw counting and detect the termination as well.

Its major advantages are as follows:

- It can be applied to the both computation models with FIFO and non-FIFO communication.
- The message complexity is typically $O(number\ of\ PEs)$.

The techniques described in this paper are applicable to many kinds of distributed processing systems.

References

[1] Rokusawa, K., Ichiyoshi, N., Chikayama, T. and Nakashima, H., "An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems," *Proc. of International Conference on Parallel Processing*, Vol.I Architecture, pp.18–22, 1988.

[2] Mattern, F., "Global quiescence detection based on credit distribution and recovery," *Inf. Proc. Lett.*, Vol.30, No.4, pp.195–200, 1989.

[3] Watson, P. and Watson, I., "An efficient garbage collection scheme for parallel computer architectures," *Proc. of Parallel Architectures and Languages Europe*, LNCS 259, Vol.II, pp.432–443, 1987.

[4] Bevan, D.I., "Distributed garbage collection using reference counting," *Parallel Computing*, Vol.9, No.2, pp.179–192, 1989.

[5] Chandy, K.M. and Misra, J., "Stability Detection," In *Parallel Program Design, A Foundation*, pp.269–288, Addison Wesley, Massachusetts, 1988.

[6] Chandy, K.M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. on Computer Systems*, Vol.4, No.1, pp.63–75, 1985.

[7] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K. and Chikayama, T., "Distributed Implementation of KL1 on the Multi-PSI/V2," *Proc. of Sixth International Conference on Logic Programming*, pp.436–451, 1989.

[8] Ueda, K. and Chikayama, T., "Design of the Kernel Language for the Parallel Inference Machine," *The Computer Journal*, Vol.33, No.6, pp.494–500, 1990.

[9] Taki, K., "The Parallel Software Research and Development Tool: Multi-PSI System," In *Programming of Future Generation Computers*, pp.411–426, Elsevier Science Publishers B.V., North Holland, 1988.