ICOT Technical Report: TR-0736

TR-0736

A Scalable Termination Detection Scheme Using Message Combining

by
N. Ichiyoshi & K. Rokusawa (Oki)

February, 1992

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
(03)3456-3191～5
Telex ICOT J32964
Minato-ku Tokyo 108 Japan

Institute for New Generation Computer Technology

A Scalable Termination Detection Scheme Using Message Combining

Nobuyuki Ichiyoshi

ichiyoshi@icot.or.jp

Institute for New Generation Computer Technology

1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

TEL: +81-3-3456-3192

Kazuaki Rokusawa

rokusawa@okilab.oki.co.jp

Systems Laboratory, Oki Electric Industry Co., Ltd.

4-11-22 Shibaura, Minato-ku, Tokyo 108, JAPAN

TEL: +81-3-3454-2111 ex. 2734

Abstract

We propose a scalable termination detection scheme for distributed computation on large-scale multicomputers. It is based on the Weighted Throw Counting technique (an application of the weighted reference counting for distributed incremental garbage collection), and employs software message combining to remove the bottleneck at the detecting process. We introduce a *delayed returning rule* for combining an unpredictable number of messages at each internal node of the combining tree. In contrast to message complexity commonly used as an efficiency criterion of termination detection algorithms, the scheme is aimed at minimizing the time complexity of detection.

Key Words: distributed computation, termination detection, weighted throw counting (WTC), message combining, time complexity

1 Introduction

Termination detection of computation is a basic functional component of parallel programs and parallel programming environments. Unlike sequential computation, termination detection may not be trivial for parallel computation because of the difficulty of obtaining a consistent global state, especially when there can be inter-processor messages in transit.

A lot of distributed termination detection algorithms have been proposed [6, 18, 12, 4]. The Weighted Throw Counting (WTC) scheme [17] is a simple and efficient algorithm based on the weighted reference counting technique for incremental distributed garbage collection [3, 21]. A similar technique was independently proposed by Mattern, who called it the "credit distribution and recovery" scheme [13]. A termination detection mechanism by WTC is realized in the implementation of a parallel logic language KL1 [20] on the Multi-PSI, a parallel computer with up to 64 processors connected by an 8×8 mesh network with worm-hole routing [14].

The original WTC scheme has a bottleneck at the detecting process; hence it is not a scalable algorithm. This paper presents a method for removing the bottleneck by software message combining. We propose a heuristic called *the delayed returning rule* for combining an unpredictable number of messages at each internal node of the combining tree, which ensures a proper combining behavior. In contrast to message complexity, a commonly used efficiency criterion for distributed termination detection algorithms, the scheme is aimed at minimizing the time complexity of detection.

2 Computational Model

Since our main concern is the time required for detecting termination of large-scale parallel computation, we would like to define distributed computation as mapped on a large-scale parallel computer — specifically, a distributed-memory parallel computer, or a multicomputer [1]. A multicomputer consists of a number of processing nodes connected by some interconnection network. Let V be the set of all processing nodes, $n = |V|$ be the number of all processing nodes, and E be the set of all pairs

$\langle N_1, N_2 \rangle \in V \times V$ such that N_1 can send messages to N_2. We assume communication path is bidirectional, that is, if $\langle N_1, N_2 \rangle \in E$, then $\langle N_2, N_1 \rangle \in E$. We require that (1) the graph (V, E) be connected, and (2) communication be reliable: every message sent will be eventually received. We do not require that the communication is FIFO, that is, we allow messages sent from N_1 to N_2 to be received out of order, but we assume that the zero-load latency of a message is predictable. The latency is the time from the start of sending a message packet at the sending node to the completion of receiving it at the receiving node. We denote the latency of a message m along the edge e by $t_m(e)$. When we talk about a spanning tree, it is one over the graph (V, E), and not (V, C) where C is the set of network channels. A message traveling along an edge in a spanning tree can cover more than one network channel.

In "second-generation" multicomputers [1] such as Symult 2010, Multi-PSI and Intel Paragon, a node can send messages to any other node by worm-hole routing, so that $E = V \times V$.

The activity of the base computation on a processing node will be referred to by a *process*.[1] We assume the following about the base computation on multicomputers.

1. A process can be in either of the two states: *active* or *idle*.

2. An active process p can send messages to any process q with $\langle N_p, N_q \rangle \in E$, where p and q reside in nodes N_p and N_q respectively.

3. An active process can spontaneously become idle at any time.

4. An idle process can never become active except when it receives a message. (For this property, a message in the base computation is sometimes called an *activation message*.)

5. Initially, all processes are idle and there are no messages; at the start of computation, one process is activated by the outside "environment" (e.g., a shell spawns the initial process).

[1] A process defined here corresponds a "process subpool" in [17]. In [17], a "process" referred to a process as defined in the language, and the collection of processes mapped on a physical processing node was referred to by a "process subpool" (in contrast to the computation as a whole which was referred to by the "process pool").

A distributed computation is terminated when all the processes are idle and there are no messages in transit (sent but not yet received). Termination is a stable state [5], that is, a terminated computation remains terminated for ever.

It is interesting to note that there can be two definitions of idleness for the same base computation. Suppose some parallel concurrent object-oriented language was implemented on a multicomputer. A process on a processor would be defined to be the collective activity of those objects that reside in that processor ("object subpool"). An object could migrate to another processor by a %migrate message[2] and create another object in another processor by a %create message. When an object needed a non-local value, it would request the value by sending a %read message and suspend. It would wake up when the required value was returned by a %value message. Now, idleness of a process could be defined either by (LT) the state where the object subpool is empty (idle $\equiv$ terminated), or by (LTS) the state where all object in the object subpool is suspended (idle $\equiv$ terminated $\vee$ suspended). The assumptions for distributed computation is satisfied in either case. Whereas termination means global termination in Definition (LT), it means global quiescence (global termination or global deadlock) in Definition (LTS). Note also that a %value message is an activation message in Definition (LTS), but not in Definition (LT).

A termination detection problem is to transform the base computation without affecting the semantics of it (typically by superposing [5]) so that the environment detects the termination of the base computation (in contrast to the superposed computation, it is also referred to by "underlying computation"). It is required that (1) when the detecting process detects termination, the base computation has indeed terminated (correctness), and that (2) the detecting process detects termination in a finite time after the base computation has become terminated (liveness).

Let P be a base program and P_t be the result of transforming P for termination detection, and T_P and T_{P_t} be the execution times of P and P_t, respectively.[3] A "good" detecting algorithm for parallel computation on a multicomputer would be

[2]In this paper, message names are marked by prefixing it with a percent sign.

[3]In general, the time T_P is not directly observable because the termination of P may not be directly observable. Indeed, the purpose of transforming the base computation P to P_t is precisely to make the termination observable.

one such that the difference $T_{P_t} - T_P$ is small compared to T_P. Thus, a "good" detecting algorithm (a) should not impose much overhead on the base computation and (b) should be able to detect termination as quickly as possible. Also, memory complexity of the algorithm should be small.

For the sake of discussion about time, we assume that there is only one system of base computation running on the multicomputer.

In section 3, we describe the WTC technique, which adds relatively small overhead to the base computation. Since the detecting process can become bottleneck as the number of processes grows, we propose a remedy in Section 4.

3 Weighted Throw Counting (WTC)

3.1 Principle

In the Weighted Throw Counting technique [17], the initially activated process is assigned a certain positive "weight" W and the environment is given the same weight, and the following invariant condition is maintained:

- Active processes and messages have positive weights.
- Idle processes have a zero weight.
- The sum of all weights of processes and messages are equal to the weight that the environment has.

Under the above condition, the condition that the environment has weight 0 is equivalent to there being no active processes and no messages in transit, that is, the base computation being terminated.

To maintain the invariant, the handling of activation messages are changed as follows. When a process sends an activation message, it splits the weight W in two positive values W_1 and W_2 such that $W = W_1 + W_2$, assigns W_1 to the activation message and retains W_2 to itself. When a process receives an activation message, the weight carried by the message is added to its weight.

To actually detect termination, a detecting process is placed on some processing node. When computation is initiated by the environment, the first process and the

detectingn process are given the same weight. The weight of the environment during computation is defined to be the weight of the detecting process minus the weights of all %return_WTC messages that are in transit. On becoming idle, a process in the base computation sends a %return_WTC (W) message to the detecting process to return the weight W of the process. When the detecting process receives a %return_WTC (W) message, it subtracts W from its weight. Note that this ensures the invariant condition on the weight.

The detecting process detects termination when its weight becomes zero. The invariant condition guarantees correctness of termination detection, and the eventual delivery of %return_WTC messages to the detecting process guarantees the liveness property.

3.2 Implementation

In actual implementations, the weight could be represented by variable length binary floating-point numbers or it could be represented by fixed length binary numbers. In the latter case, overflow and underflow of weight have to be handled. A process can simply send a %return_WTC message when the weight is overflowed. The weight underflow can be handled by the introduction of a WTC request-supply protocol.

The WTC scheme is very simple. The memory requirement is only a few words per process. Each message of the base computation is piggy-backed with the weight (also a few words). In the worst case, as many %return_WTC messages as activation messages in the base computation are sent.

In the implementation of the parallel logic language KL1 on the Multi-PSI, the system detects the termination of distributed computation (Definition LT is adopted). The weights of processes and messages are represented by 32 bit unsigned integers, and the weight of the detecting process represented by a 64 bit unsigned integer. When a process with weight 1 is to send a message, the message sending is suspended, and the process sends a %request_WTC message to the detecting process, which supplies a large fixed amount of weight (2^{24} in the current implementation). An activation message is given the weight of 2^{10} when the process sending it has the weight $W_p \geq 2^{11}$. When $1 < W_p < 2^{11}$, an activation message is given the weight

of $\lfloor W_p/2 \rfloor$. Once receiving a %supply_WTC message, a process can send more than 2^{14} activation messages. Therefore, only very few WTC supplies are usually made during computation.

4 Message Combining with Delayed Returning

4.1 Bottleneck in the Flat WTC

The WTC technique as described in Section 3 (which we will refer to as "the flat WTC") has a potential bottleneck: As the number of processes increases, the detecting process may be flooded with %return_WTC messages sent from all processes and may become bottleneck. Also, the communication channels near the detecting process may become congested.

In the KL1 implementation on the Multi-PSI, it takes the detecting process about 150 μsec to service an incoming %return_WTC message.[4] In one application (the Packing Piece Puzzle) [7], when a speedup of 50 was attained, the average execution time of a subtask was about 40 msec. This means, assuming the processor utilization of 80% (50/60 = 0.78), the detecting process is receiving one %return_WTC message per 800 μsec on the average. In this case, the detecting process was not a bottleneck. But, if the number of processors increased four fold, the detecting process would be barely catching up with the %return_WTC messages. Things would be worse with higher processor utilization and smaller subtask granularity.

4.2 Message Combining

In order to remove the bottleneck at the detecting process, it is a natural idea to put the processes in a tree structure in which each process has a small number of child processes.[5] If each process simply forwarded %return_WTC messages to the parent

[4] When a process becomes active, it notifies the detecting process (by a "%ready" message), and the latter supplies a weight 2^{24} to the process. The time 150 μsec is actually the sum of one %ready message handling time and %return_WTC message handling time, but for simplicity of argument, we ignore %ready messages here.

[5] Note, since we can identify a process with the node where it resides, we use the terminology of the tree nodes, such as "root", "child", "parent", "leaf" and "internal" (non-leaf) for describing

process, the detecting process would still be a bottleneck since it would receive the same number of messages as before. Instead, each process must *combine* multiple incoming %return_WTC messages from the child processes and send a smaller number of %return_WTC messages to the parent process. To combine a %return_WTC message at a process is to add its weight to the weight of the process. Message combining was proposed by Pfister *et al.* as a hardware mechanism for alleviating hot spot contention in multistage interconnection networks [16], and a number of variations have appeared [10, 11], including software message combining [22, 19].

4.3 Delayed Returning

The basic idea of message combining is simple, but exactly how best to combine messages depends on specific applications. Combining is straightforward when the number of messages to be combined at a process is predictable as is the case with barrier synchronization: an internal process must receive as many messages from child processes as there are child processes before it sends a combined message to its parent. In the combining of %return_WTC messages, this is not the case — it is unpredictable when and how many %return_WTC messages a process might receive. Thus, a process must somehow determine when to send a combined message up to its parent.

According to the *delayed returning rule* we propose here, the non root processes are required to keep the rate of sending %return_WTC messages lower than a certain fixed rate. The fixed rate is determined as follows. In order not to make an internal process N a bottleneck of message processing, its child processes ought not to send messages at a higher rate than the rate at which the process can handle incoming messages. Suppose the rate at which a %return_WTC message can be serviced is λ_r (the time it takes an internal process N to handle a message from a child process is $t_r = 1/\lambda_r$) and the number of the child processes is f. If a child process keeps the message sending rate λ_s to be smaller than λ_r/f, then message handling at process N will not become bottleneck. In general, for any given $0 < \rho < 1$, if t_s is chosen

the processes. In particular, the parent process of a process is not the process which spawned the latter. The set of processes does not change dynamically in our model.

so that $t_s > ft_r/\rho$, the time spent for processing %return_WTC messages at an internal process is guaranteed to be less than ρ of the total processor time of N.

The following are two ways by which the delayed returning rule is enforced:

(1) Each non-root process keeps a local clock which schedules the message sending routine at an interval of t_s. The routine sends a %return_WTC message if the process is idle when it is invoked.

(2) On becoming idle, each non-root process waits t_s before sending a %return_WTC message. If it receives an activation message before t_s elapses, it simply becomes active.

Two possible ways of how a process waits a specified amount of time is:

(a) To set an local alarm clock to wake it up after t_s unit of time, and to sleep. On receipt of an activation message during the sleep, the alarm is canceled.

(b) To go into a busy waiting loop when it becomes idle. The process counts down in the loop till the counter value reaches zero. The initial value of the counter is determined so that the time t_s elapses in the countdown. On receipt of an activation message during the sleep, the busy loop is exited.

An idle process that has not sent a %return_WTC message after it has become idle for the last time is said to be in the *delay slot*. The WTC invariant condition is changed to allow an idle process in the delay slot to have a positive weight. The life time of a process appears to become longer to the environment. An internal process treats a %return_WTC that arrives at it just as an activation message bringing a null task. The weight of the message is added to that of the process, and if the process is idle when it receives the message, it behaves as if it became active and immediately became idle again.

5 Message and Time Complexities

5.1 Message Complexity

Let M be the set of all activation messages generated and $m = |M|$, R be the set of all %return_WTC messages generated and $r = |R|$. In the flat (i.e., one without combining tree) WTC with or without delayed returning, at most one %return_WTC message is sent for one activation message. Thus, $r \leq m$.

As for the combining WTC, let us suppose that the fanout of the combining tree is constant $f(> 1)$. In the worst case, every activation message is received by an idle leaf process and cause d %return_WTC messages to be sent, where $d = \log_f n$ is the depth of the combining tree. Thus, the worst case message complexity is $O(m \log n)$, slightly worse than $O(m)$.

In a typical case, however, activation messages may arrive at active processes. Moreover, with the combining WTC, an idle process in the delay slot may receive an activation message. This causes a %return_WTC message which would have been sent in the flat WTC to be canceled. A process could become active and idle alternatively a number of times, but could send only a small number of %return_WTC messages. Note this effect is due to delayed returning. If the flat WTC adopted delayed returning alone (the (flat) WTC with delayed returning), it would enjoy the effect.

5.2 Time Complexity

5.2.1 Flat WTC

In general cases, processes can become active and idle alternatively many times during computation. Suppose that at some stable stage in the computation, each process becomes active at the rate of λ_a times per unit time. In the flat WTC, %return_WTC messages will be generated at the rate λ_a per process. Unless $n\lambda_a$ is less than a unit time, the detecting process cannot catch up with the incoming %return_WTC messages. To prevent the detecting process from becoming a bottleneck, λ_a has to decrease proportionally to $1/n$ — in other words, the average granularity of ("subtask") has to increase proportionally to n (or local idle time has to increase).

5.2.2 Flat WTC with Delayed Returning

The above mentioned bottleneck can be removed by decreasing the generation rate of %return_WTC messages. The delayed returning rule ensures that the sending rate be kept less than $\lambda_r/(n-1)$ (the fanout at the root is $n-1$), where λ_r is the rate at which %return_WTC messages can be serviced.

If the maximum message latency of the network is $O(n)$, the time it takes to detect the termination is $O(n)$. This upper bound is attained when $\Theta(n)$ processes participate in the base computation in the last $O(n)$ time of the base computation, i.e. when the parallel computer is efficiently used. Thus, the observable runtime T_{P_t} is at least $\Omega(n)$. If the base computation time T_P is $o(n)$, the detection time will eventually become bottleneck as $n \to \infty$. However, since the service time of a %return_WTC message is very short, this is not expected be a problem in practice. Again, In the case of the KL1 implementation, even with $n = 10,000$ processors, the total service time of 10,000 %return_WTC messages is only 1.5 seconds. The runtime of a large-scale application program that makes an efficient use of the large-scale parallel computer would be much much longer. (The total amount of computation necessary for maintaining a constant efficiency usually has to increase more than linearly in the number of processors [9]. This means the parallel runtime has to grow as the number of processors grows.)

5.2.3 Combining WTC

A theoretically better detection time can be achieved by employing a spanning tree for message combining. We consider the case that all processes (or $\Theta(n)$ processes) participate in the base computation. Under the constant fanout f, the detection time will be $\Omega(\log n)$. Since combining of WTC from all processes is a special case of single-node accumulation, the detection time is also Ω(single-node broadcast time).

The lower bound can be attained in a binary hypercube, since a two-rooted binary balanced tree can be embedded in it [2].

As for a two-dimensional square mesh network, the lower bound of $\Omega(\sqrt{n})$ can be attained. A very simple spanning tree with which to achieve this bound is as follows. Let processing nodes be identified by the coordinate $\langle x, y \rangle$ $(0 \leq x, y \leq$

$\sqrt{n}-1$). The tree has the root at $\langle 0,0\rangle$. The child nodes of the root node are $\langle x,0\rangle$ $(0 \leq x \leq \sqrt{n}-1)$. The child nodes of node $\langle x,0\rangle$ are $\langle x,y\rangle$ $(0 \leq y \leq \sqrt{n}-1)$. Thus, the depth of the tree is 2, the fanout of all internal nodes except the root node is $\sqrt{n}-1$, and the fanout of the root node is $2(\sqrt{n}-1)$. The maximum latency of %return_WTC messages is $\Theta(\sqrt{n})$. Note, although there could be contentions between the messages, that would not aggravate the time complexity. For example, $\sqrt{n}-1$ %return_WTC messages could simultaneously compete for the network channel between $\langle x,0\rangle$ and $\langle x,1\rangle$. But since it takes the processing node $\langle x,0\rangle$ an $\Theta(\sqrt{n})$ time to process these messages, the contention does not affect the order of the overall time for termination detection. Alternatively, by recursively dividing the square mesh into four square sub-meshes and placing the root at the center of each sub-mesh, a spanning tree with fanout 4 and depth $\lceil \log_4 n \rceil$ can be constructed. The distance between the global leaf node and the global root node along the tree edges is $\Theta(\sqrt{n})$. This spanning tree also attains the lower bound of detection time. Some channels are on more than one tree edge, but since the degree of this "overloading" is at most 2 (under the e-cube routing — to route in the first dimension then in the second dimension), the contention can only increase the detection time by a constant factor.

6 Discussions

A number of distributed termination detection schemes have been published, such as [6], [18], [12], and [4]. But as far as the authors are aware, complexity arguments are concentrated on the message complexity [18, 4] with time complexity ignored or only lightly touched upon. Probably, the reason is that most termination detection schemes are intended for very general distributed environments where it is hard to make assumptions about node-to-node or site to site communication delay. In contrast, the scheme proposed in the current paper is specifically intended for use in large-scale parallel computers.

Besides detecting algorithms, there are algorithms for testing the termination. Some distributed termination schemes repeatedly use such an algorithm and report the termination when after a number of failures the termination test becomes

successful. Mattern [12] compares and discusses such algorithms. Typically, each process maintains a counter that records the number of messages sent minus the number of messages received, and reports the value to the testing process when requested. The problem is that, since the reporting times can differ from process to process, the testing process does not in general obtain a consistent global state. This is solved by introducing local flags or message generations, etc. By using a spanning tree for broadcasting the request for report and for combining the reports[6], the time required for a test can be minimized. The main difference is that one global test can be initiated for one active message. Thus, when only a limited number of processors are in use, the repeated testing scheme will generate much more control messages than the WTC scheme. The worst-case message complexity $r = \Omega(mn)$. The advantage over the WTC scheme is that message counting is much simpler than the handling of weight. In particular, active messages do not have to be piggy-backed with a weight.

A number of message combining schemes have been proposed. In Pfister *et al.* the network switch has a buffer at the input to hold those messages not yet served, and on arrival of a new input message the buffer is looked up for combinable messages. The congestion level of the switch determines the level of combining. In contrast, the delayed returning rule removes the bottleneck at the receiving node by limiting the message generating rate at the message source. The concept of *window* [19] is close to the sending interval t_s of the delayed returning. But [19] does not indicate the right size of the window or the way to guarantee the eventual message sending to the parent. The problem with the delayed returning is a relatively high overhead of delaying a message sending, and this may confine its use in limited situations, such as termination detection.

7 Conclusions

A termination detection scheme for use in large-scale multicomputers was presented. It is based on the Weighted Throw Counting (WTC) technique, but the bottleneck

[6] In order to reduce the number of tests, the sending of the combined message to the parent should be delayed until the process becomes idle.

at the detecting process is removed by message combining. The delayed returning rule guarantees a proper combining behavior at a combining node in the presence of unpredictable number of incoming messages. It also helps to greatly reduce the number of control messages (%return_WTC messages). The time complexity was discussed.

References

[1] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–25, 1988.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation — Numerical Methods.* Prentice Hall, 1989.

[3] D. I. Bevan. Distributed garbage collection using reference counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176–187, June 1987. Also in *Parallel Computing*, Vol.9, No.2, pp.179–192, 1989.

[4] S. Chandrasekaran and S. Venkatesan. A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, 8(3):245–252, 1990.

[5] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] E. W. Dijkstra and C. S. Sholten. Tremination detection for diffusing computations. *Information Processing Letters*, 16(5):217–219, 1980.

[7] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the Multi-PSI. In *Proceedings of PPoPP'90*, pages 50–59, 1990.

[8] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine (PIM) architecture. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 208–229, 1988.

[9] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures: A survey. In *Proceedings of The 1991 International Conference on Supercomputing*, 1991.

[10] T. Lang and L. Kurisaki. Nonuniform traffic spots (NUTS) in multistage interconnection networks. *Journal of Parallel and Distributed Computing*, 10(1):55–67, 1990.

[11] R. L. Lesher and M. J. Thazhuthavetil. Hotspot contention in non-blocking multistage interconnection networks. In *Proceedings of ICPP'90: Volume I*, pages 401–404, 1990.

[12] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[13] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(4):195 200, February 1989.

[14] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 436–451, 1989.

[15] H. Nakashima, K. Nakajima, S. Kondo, and Y. Takeda. Architecture and implementation of PIM/m. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, 1992. To appear.

[16] G. F. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[17] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. I Architecture*, pages 18–22, 1988.

[18] N. Shavit and N. Frances. A new approach to detection of locally indicative stability. In L. Kott, editor, *Proceedings of International Conference on Automata, Languages and Programming*, pages 344–358, 1986. Lecture Notes in Computer Science 226, Springer.

[19] P. Tang and P.-C. Yew. Software combining algorithms for distributed hot-spot addressing. *Journal of Parallel and Distributed Computing*, 10(2):130–139, 1990.

[20] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494 500, 1990.

[21] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432–443, June 1987.

[22] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.