Termination Detection: Models and Algorithms for SPMD Computing Paradigms

Erturk Kocalar and Ashfaq Khokhar
Dept. of Electr. and Comp. Eng.
Univ. of Delaware, Newark, DE 19716
{kocalar,ashfaq}@eecis.udel.edu

Susanne E. Hambrusch
Dept. of Computer Sciences
Purdue Univ., W. Lafayette, IN 47907
seh@cs.purdue.edu

Abstract

We consider two termination detection models arising in parallel applications, the APS and the AP model. In the AP model, processors are either active or passive and a passive processor can be made active. In the APS model, processors can also be in a server state. A passive processor can enter the server state, but does not become active again. We describe and analyze algorithms for both models and present experimental work highlighting the differences between the models. We show that in almost all situations the use of an AP algorithm to detect termination in an APS environment will result in a loss of performance. Our experimental work on the Cray T3E provides insight into where and why this performance loss occurs.

1 INTRODUCTION

Termination detection is a fundamental communication operation in data-parallel programs, especially programs in the Single Program Multiple Data (SPMD) mode. Synchronization may be needed during as well as at the end of program segments. When synchronization only needs to ensure that no processor (PE) advances beyond a certain point and each PE knows in advance when it has finished its computations, barrier-style synchronization primitives can be used [3, 10]. However, in many applications synchronization points cannot be explicitly or statically placed in the program. This happens, for example, when synchronization events are data-driven. Termination detection achieves global synchronization for such scenarios.

A major challenge of termination detection (TD) is to repeatedly capture a snapshot of the global state of the system in order to detect termination as soon as possible. Termination should be detected without creating communication bottlenecks and without destroying an existing balance between communication and computation.

Termination detection has been extensively studied in the realm of distributed processing [2, 4, 5, 8, 9, 13, 14, 15]. Solutions for parallel environments are described in [16, 17]. TD solutions developed for distributed systems may make assumptions not directly applicable to a parallel system. Termination detection is generally achieved by sending control messages (TD messages). These are in addition to the work messages, i.e. the messages sent by the original program. In a parallel system, TD messages can destroy a carefully achieved balance between communication and computation. Our goal is to develop TD algorithms for parallel systems that minimize the number of control messages as well as the amount of additional delays induced by acknowledgments of work messages.

Following conventional terminology used in termination detection, we assume that a processor is either active or passive. Whether processors start out as active or passive is not crucial to our algorithms. In our underlying applications processors start out as active. An active processor performs local computations and it can send remote work requests to other processors. When an active processor has finished its assigned work load, it becomes passive and waits for termination to occur. A passive processor can send and receive control messages. When it receives remote work requests its status changes. *Termination occurs when all the processors are passive and there is no message in transit.*

In this paper we investigate two TD scenarios which differ on how passive processors respond to work messages. Assume the application requiring TD executes on parallel system consisting of N processors (PEs). We do not make any assumptions about the underlying interconnection network. We only assume that the PEs communicate using a set of communication channels. The first TD model is the general environment in which a passive processor, upon receiving a

work message, becomes active again and is capable of generating new work messages. We refer to this scenario as the active-passive (AP) model. A number of parallel applications requiring termination handle incoming work requests in a different way [6, 7, 12]. In such applications, when a passive processor receives a work request from another processor, the work request obeys locality. By that we mean that in order to process and complete the work request, no communication is required and all the data needed is available locally at the processor. A passive processor receiving a work message does thus not become active. It can be viewed as being in a "server state." We refer to this termination scenario as the active-passive-server (APS) model. Figure 1 shows the possible transitions between states for the APS and AP model as well as barrier synchronization.

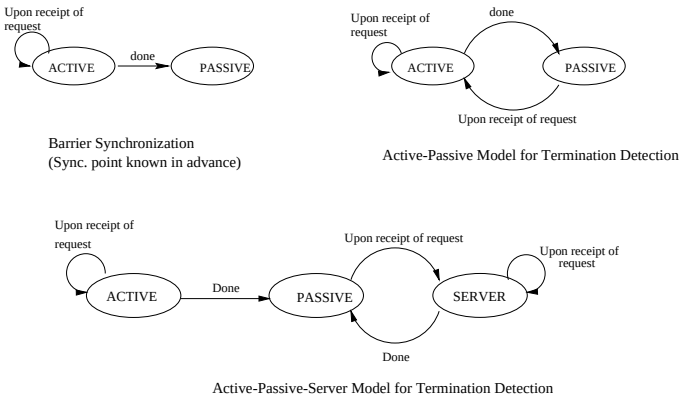


Figure 1: Synchronization models

We present and analyze efficient TD algorithms for the APS and the AP model. Clearly, any algorithm for the AP model detects termination in the APS model. Our work focuses on the loss in performance when an algorithm for the AP model is used for the APS model. Two of the performance measures we consider are *termination delay*, i.e. the difference between the time when global termination occurs and the time it is detected, and *slowdown*, i.e. the increase in execution time due to the presence of control messages and congestion. We present complexity analysis showing how the use of an AP algorithm in an APS scenario results in an increase in control messages. Our experimental results echo the complexity results. They show that in almost all situations the use of an AP algorithm to detect termination in an APS environment results in significant performance loss.

2 RING ALGORITHMS

In this section we describe TD algorithms when processors use a token-passing logical ring network on processors $PE_0, \ldots, PE_{N-1}$. Only control messages use the ring structure.

Ring Algorithm for the APS Model:

Assume that initially every processor is active. When a processor becomes passive and initiates a check for termination, it generates a clean token to be passed around the ring. A passive processor receiving a clean token leaves the token clean. An active or server processor receiving a clean token makes the token dirty. A processor receiving a dirty token does not change it, but passes it on. When the token returns to its initiator and it is dirty, the initiator knows that there was at least one non-passive PE in the system. A passive initiator sends out another clean token.

Concluding that termination has actually occurred requires not only the receipt of a clean token, but the knowledge that all the messages sent have been received. To achieve this, the token is augmented with two fields that keep track of the number of sends and receives, nr_send and nr_rec, respectively. When the initiator receives a clean token and nr_send $\neq$ nr_rec, a new clean token is generated. This process is repeated until the initiator receives a clean token and the total number of sends and receives issued by all PEs is equal.

Ring Algorithm for the AP Model:

The algorithm described for the APS models fails to detect termination for the AP model. More precisely, it could conclude that termination occurred when the system has not terminated. The problem lies in active processors now being able to make passive processors active and these processors can send messages which make other passive processors active.

In the ring algorithm for the AP model every processor uses, in addition to nr_send and nr_rec, a message flag, m_flag. The flag is initially set to 0. A processor PE_i sets the flag to 1 whenever it sends a work message to another processor. PE_i resets m_flag to 0 when it receives a clean or dirty token. When a clean token reaches a passive processor and that processor's m_flag is set to 1, the token becomes dirty. Adding the above described actions to the algorithm for the APS model results in a correct algorithm for the AP model when there is only one token present in the ring. If two or more PEs become initiators, they will each inject a clean token into the system. We solve this contention by assigning priorities to processors (PEs of higher priority kill tokens generated by the PE with lower priority). When multiple tokens are present, a

token with lower priority can reset m_flag in PE_i and later be purged. Another token of higher priority can arrive at PE_i later and find no evidence that PE_i sent a work message which made another processor active. We solve this problem by recording the priority of the processor resetting m_flag. When a clean token passes through a passive processor with m_flag = 0 and the priority of the token is higher than the recorded priority, the token is made dirty and the priority is updated. When a processor sends a work message and sets m_flag =1, it sets the priority as being undefined. As holds for all TD algorithms described in this paper, correctness arguments are omitted due to space constraints and will appear in the full version [11].

3 TREE ALGORITHMS

In this section, the TD algorithms use logical tree structure for TD messages. Processors becoming passive send TD information either up the tree to a parent PE or to a designated PE on a level close to the root, a PE on the *control level*.

In the logical tree every leaf corresponds to one of the N processors. PEs are grouped together in groups and one of the PEs within each group is made the parent of the group. These selected parent PEs form the second level of the tree. The third level is constructed by grouping these 2nd level parents and selecting 3rd level parents (a processor can be a parent at multiple levels).

Tree Algorithm for the APS Model:

Assume that every processor knows its level, its parent, and its number of children in the logical tree. Further, each PE is assigned one PE on the control level. In Algorithm Tree_APS, an active PE becoming passive for the first time informs its immediate parent by sending a TD message which contains the number of sends and receives it has issued so far. When all the PEs in a group have reported to their parent PE, the parent PE combines the information about sends and receives issued within the group and sends this information up to its parent. Likewise, at the upper levels of the tree, each parent sends the information upwards when it has received the information from all of its group members. After each child of the root has reported to the root and the root has become passive, the root checks whether the number of sends equals the number of receives. If they are, the root declares termination. If the number of sends and receives at the root do not match, there are work-request messages in transit and thus the root does not issue termination.

A passive PE goes into the server state if an active PE sends it a work request. When a PE becomes passive after being in the server state, it does not send a control message to its parent, but it reports directly to the assigned processor in the control level. If this processor at the control level has not yet received reports from all its children, it simply records the received information. Otherwise, it passes the received information up towards the root. Eventually, the number of sends and receives will match and the root detects termination. The choice of the control level can have significant impact on the performance. A control level close to the root implies that control messages traverse fewer hops, but it increases the potential for communication bottlenecks.

Tree Algorithms for the AP Model:

The first AP algorithm, **Algorithm Tree_CNTR**, can be viewed as an extension of the APS tree algorithm. Similar to Tree-APS, we use a control level. We further maintain for each processor an array $Counter$ of size N which is used to record the number of messages sent to each individual processor. When an active processor PE_i sends a work message to PE_j, PE_i increments $Counter_i[j]$. PE_j, after receiving a work message, decrements its $Counter_j[j]$.

When active PE_i becomes passive for the first time, it sends a TD message containing array $Counter_i$ to its parent and PE_i reinitializes array $Counter_i$ to zero. Upon receiving TD messages from its children, a PE merges the arrays received. Once the node has received TD messages from all its children, it sends a TD message consisting of the updated array to its parent. This process continues until the root receives the TD message. When an active processor PE_i becomes passive any time other than the first time, PE_i sends a TD message with the new values in array $Counter_i$ to its assigned processor in the control level.

The root declares termination when (i) it has received at least one TD message from every child, and (ii) every entry in its $Counter$-array is zero. The flow for TD messages is analogous to the flow in the Tree-APS. However, TD messages contain now an array of size N.

The second AP tree algorithm, **Algorithm Tree_ACK**, operates on a tree without using the control level. This algorithm is based on a well-known approach for termination detection [1]. Algorithm Tree_ACK can be viewed as an algorithm with static and dynamic parental responsibility. Termination is detected by processors sending TD messages from the leaves towards the root of the static, logical tree. Once the root has received a TD message from each child and is passive itself, it declares termination. Hence,

communication of the tree consists of a single flow from the leaves towards the root. This is made possible by using a message-based dynamic parental responsibility which is established between an active and a passive processor. This new type of responsibility makes use of acknowledgement messages and may cause delays in termination detection. We refer to [1] for more details.

Complexity Comparisons:

In this section we present an analytical comparison of the three TD tree algorithms. The asynchronous nature of termination detection coupled with the difficulty in capturing congestion in a parallel system makes general analysis difficult. We thus concentrate on situations which we judge to be representative of scenarios arising in APS applications. We assume that the underlying computations occur in the form of a *diffusing process*. Such a process involves n of the N processors, $n \leq N$. At the beginning of a diffusing process, one of the n PEs is active. The other $n-1$ PEs are passive. The active processor sends work messages to the passive processors. We assume that a passive processor receives at least one work message and that a total of m work messages is sent. In an actual application, several diffusing processes may exist simultaneously. We analyze the best and worst case situation in terms of four parameters:

- TD_tot_msgs: the total number of TD messages sent
- TD_dly_msgs: the number of TD messages sent after the last processor becomes passive
- TD_tot_comp: the total amount of additional work incurred by the algorithm during a diffusing process
- TD_dly_comp: the amount of work done by all the processors to detect termination after the last processor becomes passive.

BEST CASE			
	APS	CNTR	ACK
TD_tot_msgs	$h+n+2^c$	$h+n+2^c$	$m+h$
TD_dly_msgs	c	c	h
TD_tot_comp	$O(n)$	$O(nN)$	$O(m)$
TD_dly_comp	$O(c)$	$O(cN)$	$O(h)$
WORST CASE			
	APS	CNTR	ACK
TD_tot_msgs	$h+m+cm$	$h+m+cm$	$m+h$
TD_dly_msgs	$n+2^c$	$n+2^c$	$m+h$
TD_tot_comp	$O(m)$	$O(mN)$	$O(m+h)$
TD_dly_comp	$O(n)$	$O(nN)$	$O(m+h)$

Figure 2: Complexity analysis of different tree algorithms under best case and worst case scenarios

The best case for a diffusing process generally occurs when the passive processors become servers exactly once and server processors finish at times minimizing the four quantities stated above. The worst case for a diffusing process occurs when processors switch between passive and server state often and every new passive state results in a maximum number of new control messages. Figures 2 summarize the bounds for the three algorithms. Besides the quantities n, N, and m already defined, we use h as the height of the tree and c as the position of the control level. Note that $c < h$ and $2^c << N$.

A complete discussion on how the bounds are obtained will appear in the full version of the paper. We point out two features which are crucial to understanding the experimental results. Algorithms Tree_APS and Tree_Counter send basically the same number of control messages. Their difference lies in the size of the control messages and the computation required. Algorithm Tree_ACK does perform more computations compared to Tree_APS, but the difference is not striking. What sets Tree_ACK apart from Tree_APS is the fact that every work message induces two control messages. In both the best and the worst case scenario of a diffusion process the difference in the number of control messages is significant. The experimental results reflect these differences.

4 EXPERIMENTAL RESULTS

In this section we discuss the performance of the ring- and tree-based TD algorithms on a Cray T3E using MPI message passing primitives.[1] The focus of our experimental work is on demonstrating that the use of a termination detection algorithm designed for the AP model can result in considerable performance loss for applications that exhibit APS characteristics.

The TD algorithms have been executed under a number of different scenarios. The experimental results we include in this paper are for the scenario when processors are initially assigned work loads based on a Poisson distribution. We use Poisson mean values of 1, 2, 5, 20 and 40. A mean value of 1 implies that initially almost all PEs get similar work loads and thus finish very close to each other. The higher mean values imply increasing disparity among the initial work loads at different processors. Within each processor, the work load consists of local and remote work. The percentage of local and remote work is a parameter. We report results for 1%, 2%, and 5% of the work in each processor being remote work requests to other

[1] Results reported are for machine size of 128 PEs, unless otherwise stated.

processors. When a remote work request is sent out, all other processors are equally likely to be the destination. And for each remote work request, the probability of it waiting for an acknowledgment is 0.5. For the remote requests that must be acknowledged, the requesting processor waits for the acknowledgment before resuming its local work.

Our performance study focuses on two measures *termination delay* and *slowdown*, defined in Section 1. Figure 3 compares the termination delay for three ring algorithms on various machine sizes. Two of these ring algorithms are explained in Section 2. We refer to them as algorithms Ring_APS and Ring_FLAG. In addition, we include a ring algorithm for the AP model, algorithm Ring_COUNTER, which does not use flags, but a counter array of size N is used in every processor. The use and function of the Counter entries is as described for Algorithm Tree_CNTR in Section 3.

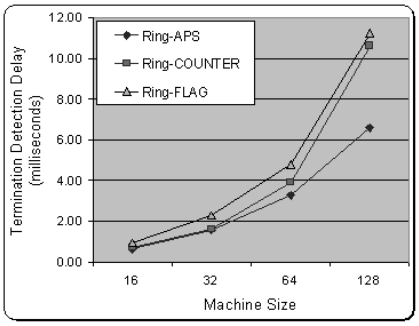


Figure 3: TD delay for three ring algorithms.

The termination delay in Figure 3 corresponds to the average termination delay over different computation loads and remote work requests. As predicted, for all three ring algorithms, the termination delay increases linearly with the increase in number of processors. For larger machine sizes, the algorithms Ring_FLAG and Ring_COUNTER perform worse than Ring_APS. The reason is that Algorithm Ring_FLAG potentially makes one additional pass through the ring and Algorithm Ring_COUNTER suffers due to large token size which is proportional to the number of processors.

Next, we compare Ring_APS with the tree-based algorithms. Figure 4(a) compares the termination detection delay of Ring_APS and tree based algorithms for different machine sizes. The tree based algorithms, Tree_APS and Tree_ACK, outperform Ring_APS. The Ring_APS algorithm suffers due to its inherent sequential flow of the token through the processors. Figure 4(b) gives a closer look at the performance of the tree-based algorithms. The performance curves correspond to a workload scenario of mean=20 and remote workload=5%. In terms of TD delay, the performance of the Tree_APS and the Tree_ACK is not strikingly different. On the other hand, the effect of large message sizes for a token is seen in the termination detection delay of Tree_CNTR.

Figure 5(a) captures the slowdown in terms of total TD messages. Algorithm Tree_ACK creates over twice the number of TD messages compared to Tree_APS and Tree_CNTR. This is also true for different load distributions as shown in Figure 5(b). The number of TD messages generated by algorithm Tree_CNTR is close to the number of TD messages generated by Tree_APS. This is expected, because the two algorithms only differ in the size of the control messages: Algorithm Tree_APS uses a $O(1)$ size token whereas Tree_CNTR uses $O(N)$ size token. In our experiments, we have observed that the algorithm Tree_CNTR slows down the underlying computation significantly. This also results in a higher termination detection delay as shown in Figure 4(b). Note that Tree_APS algorithm outperforms Tree_ACK and Tree_CNTR algorithms in APS environment, both in terms of termination detection delay and overall computation slowdown. For additional discussion on the performance of the TD algorithms we refer to [11].

5 CONCLUSION

We considered two models for termination detection arising in SPMD applications, the APS model and the AP model. For each model we described a number of different algorithms which adapt and modify methods for distributed termination detection algorithms to parallel systems. Our experimental results show that under almost all circumstances the use of an AP algorithm for an APS scenario results in a considerable loss of performance.

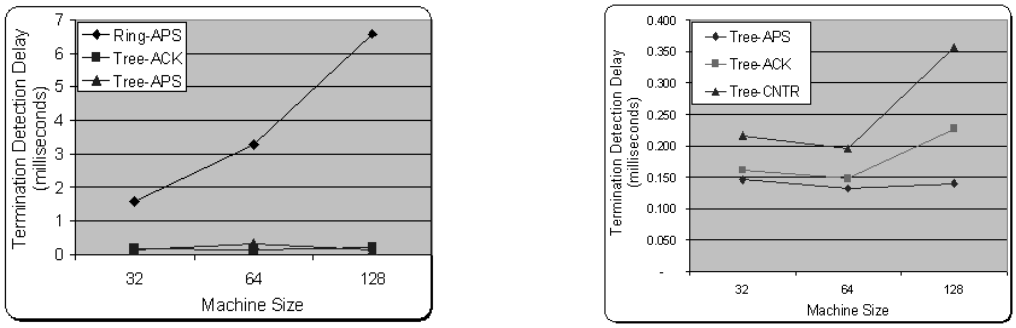
References

[1] D.P. Bertsekas and J.N. Tsitsklis. *Parallel and Distributed Computation.* Prentice Hall, 1989.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distr. systems. *ACM Trans. on Comp. Systems*, 3(1):63–75, 1985.

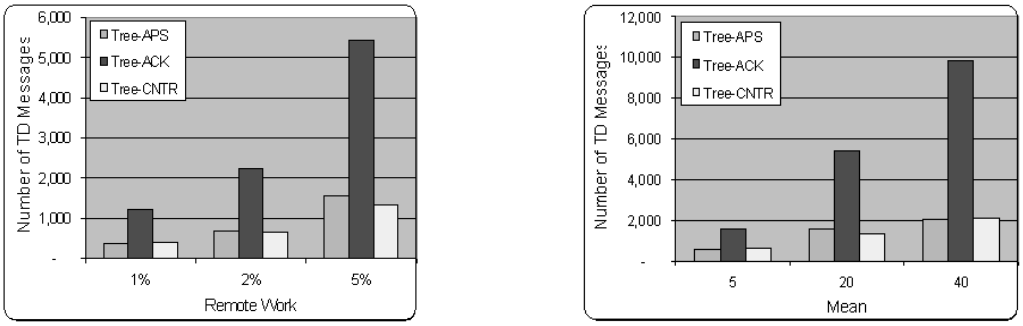
[3] H.G. Dietz, M.T. O'Keefe, and A. Zaafrani. Static scheduling for barrier MIMD architectures. *The Jour. of Supercomp.*, 5(4):263–289, 1992.

[4] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection al-



(a) (b)

Figure 4: TD delay for (a) Ring_APS and two tree algorithms (b) Comparing tree algorithms.



(a) (b)

Figure 5: Total number of TD messages (a) varying remote workload with mean=20 (b) varying mean with 5% remote work request per processor on 64-processors

gorithm for distributed computation. *IPL*, 16(5):217–219, 1983.

[5] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. on Soft. Eng.*, 8(3):287–292, 1982.

[6] A.Y. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the Barnes-Hut method for n-body simulation. In *Supercomp. '94*, 1994.

[7] S. E. Hambrusch and A. Khokhar. Maintaining spatial data sets in distr.-memory machines. In *Proc. of the Eleventh Inter. Paral. Proc. Symp.*, 1997.

[8] C. Hazari and H. Zedan. A distributed algorithm for distributed termination. *IPL*, 25(5):293–297, 1987.

[9] S. Huang. A fully distributed termination detection scheme. *IPL*, 29:13–18, 1988.

[10] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Inter. Conf. on Paral. Arch's and Compilation Tech's*, 1994.

[11] E. Kocalar, A. Khokhar, and S.E. Hambrusch. Termination detection: Models and algorithms for SPMD computing paradigms. Technical Report, 1999, `http://www.cs.purdue.edu/homes/seh/term.ps`.

[12] K.L. Ma and T.W. Crockett. A scalable parallel cell-projection volume rendering algorithms for 3-dim. unstructured data. In J. Painter, G. Stoll, and K.-L. Ma, editors, *IEEE Parallel Rendering Symp.*, pp 95–104, 1997.

[13] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *Jour. of Systems and Soft.*, 1997.

[14] F. Mattern. Algorithms for distributed termination detection. *Distr. Comp.*, 2:161–175, 1987.

[15] G. Tel. *Introduction to Distributed Algorithms.* Cambridge Univ. Press, 1994.

[16] R. Wisniewski, L. Kontothanassis, and M. L. Scott. High performance synch. algorithms for multiprogrammed multiprocessors. *Proc. Practices of Paral. Prog. Conf.*, pp 199–206, 1995.

[17] C. Xu and F.C.M. Lau. Efficient termination detection for loosely synchronous applications in multicomputers. *IEEE TPDS*, 7(5):537–544, 1996.