TERMINATION DETECTION IN DATA-DRIVEN PARALLEL COMPUTATIONS/APPLICATIONS\textsuperscript{1,2}

Ashfaq A. Khokhar
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, IL 60607
ashfaq@eecs.uic.edu

Susanne E. Hambrusch
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
seh@cs.purdue.edu

Erturk Kocalar
WebTV Networks
Mountain View, CA 94043
erturkk@corp.webtv.net

Abstract
High performance computing applications with data-driven communication and computation characteristics require synchronization routines in the form of eureka, barrier, or termination synchronization. In this paper we consider termination synchronization for two different execution models, the AP and the APS model. In the AP model, processors are either active or passive and a passive processor can be made active by another active processor. In the APS model, processors can also be in a server state. A passive processor entering the server state does not become active again. In addition, a server processor cannot change the status of other processors. We describe and analyze solutions 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 loss of performance. Our experimental work on the Cray T3E provides insight into where and why this performance loss occurs.

Keywords: Data-driven algorithms, termination detection, tree and ring networks, synchronization.

\textsuperscript{1} A preliminary version of this paper has appeared in the Proceeding of the International Conference on Parallel and Distributed Computing Systems, 1999.
\textsuperscript{2} Work supported in part by the National Science Foundation Grants 9875662-CCR and 9988339-CCR.
1. INTRODUCTION

High performance computing applications with irregular communication and computation characteristics require synchronization routines in the form of eureka, barrier, or termination synchronization [14]. Of these, termination synchronization, also referred to as termination detection (TD), is the most difficult and costly routine since it requires repeatedly capturing the global state of parallel or distributed system. TD dynamically determines whether all processors have reached the end of the execution of their program segment. Unlike static synchronization routines such as eureka and barrier, it cannot be incorporated during coding or at compile time nor can it be replaced by these static routines. Given the overhead introduced, it is crucial to understand the communication patterns induced by TD routines under different execution scenarios and their impact on the overall performance of the parallel program. Termination detection is often needed in data-driven applications like real-time multimedia, interactive visualization, data mining, ray tracing, and event-driven simulation. In this paper we study termination detection for two different execution models arising in data-driven applications. The two models differ on how processors respond to incoming work requests. We develop and analyze algorithms based on different approaches for both of these models and present experimental work highlighting the differences between the models.

In the execution model assumed by earlier termination detection work for distributed processing paradigms [5], processors are active and enter a passive state after having completed the assigned work. Active processors can send messages to other processors. We refer to these messages as work messages. A passive processor receiving a work message becomes active and thus potentially can make other passive processors active again. Termination is achieved only when all the processors are passive and there are no messages in transit. We refer to this program execution framework as the AP (Active-Passive) model. Existing distributed TD algorithms for this model employ techniques that repeatedly capture the state of all the processors by sending control messages (TD messages). These messages tend to be short and are in addition to the work messages embedded in the application. TD messages have the potential to alter an existing balance between communication and computation in an algorithm.

On the other hand, in execution models assumed by a number of parallel applications requiring termination detection, processors respond to work messages differently from what is captured in the AP-model. One such scenario is that of work messages obeying locality [7,9,11]. Locality means that in order for a processor to complete the work requested in a work message, no further communication is required and needed data is available locally at the destination processor. Under this scenario, passive processors receiving a work message do not become “active.” Instead, they can be viewed as being in a “server state,” and will not make other passive processors active again. We refer to this execution model as the active-passive-server (APS) model. This seemingly subtle difference can have a significant impact on the performance of a TD algorithm.

In this paper we present and analyze efficient termination detection algorithms for both the AP and the APS models. Our TD algorithms make no assumption about the underlying interconnection network. We assume a distributed-memory environment, with every processor assigned an initial workload. The TD algorithms use the communication pattern of a ring or tree structure, two often employed communication patterns for termination detection. A ring communication pattern is used for its simplicity, but tends not give the best performance due to the large diameter of a ring. A tree or tree-like communication pattern is a natural and generally more efficient pattern for detecting termination. Clearly, any algorithm for the AP model correctly detects termination in the APS environment. We discuss where and how the use of an AP algorithm in an APS environment results in an increase in control messages and an increase in the work performed by the processors. For selected cost measures, we present a complexity analysis. For experimental purposes we use a probabilistic application model to generate synthetic workloads with diverse characteristics to simulate the application environments for AP and APS models. These experimental results verify our theoretical analysis and show that in almost all situations the use of an AP
algorithm to detect termination in an APS environment results in significant performance loss. For machine sizes of 64 processors and up, the loss on the Cray T3E is a factor of 2-4.

The contributions and originality of the work presented in this paper are summarized below:

- Analysis of existing AP termination detection algorithms for the APS execution model.
- Development of efficient algorithms both for AP and APS models considering a distributed-memory parallel computing framework.
- Theoretical and experimental analysis of the proposed TD algorithms.

The paper is organized as follows. In Section 2, we further discuss the AP and APS models and sketch existing work. Section 3 compares termination detection algorithms for both models when TD messages use an underlying logical ring structure for communication. Section 4 describes and compares TD algorithms based on tree-like communication. Section 5 describes the implementations of the proposed algorithms on the Cray T3E and presents experimental results. Conclusions are presented in Section 6.

2. TERMINATION DETECTION

Termination detection (TD) has been extensively studied in the realm of distributed processing [4,5,6,12,13,17]; solutions for parallel environments are described in [1,10,20,21]. Termination detection solutions developed for distributed systems may make assumptions not directly applicable to a parallel system. These include assuming that the sending and receiving of a message corresponds to one atomic action [16], that message sending is blocking (i.e., every send is matched by a corresponding receive operations and is thus acknowledged) [21], or that messages arrive in the order they were sent [3]. Incorporating such assumptions to an existing parallel program can lead to a significant deterioration in performance.

Assume the application requiring termination detection executes on a parallel system consisting of $N$ processors. Throughout, we do not make any assumptions about the underlying interconnection network. We only assume that the processors communicate with each other using a set of communication channels. As already stated, we assume that initially every processor is assigned a workload and is thus active. An active processor can send work messages to other processors. An active processor becomes passive when it has finished its assigned job. The functionality of a passive processor in the two models is different. In both models, passive processors can send and receive control messages. In the APS model, a passive processor, upon receiving a work message, enters the server state. Processors in the server state do not send work messages to other processors. Hence, they cannot change the state of other processors. In the AP model, a passive processor, upon receiving a work message, becomes an active processor and may send remote work requests to other processors. Figure 1 shows the possible transitions between states for the APS and AP model as well as barrier synchronization.

The overall goals of a termination detection algorithm can be summarized as follows:

1. Minimize the termination delay; i.e., the difference between the time when global termination occurs and the time it is detected.
2. Minimize the slowdown; i.e., the increase in time due to the presence of control messages and the arising congestion in the system.
3. Make no assumption about the underlying network topology or the type of communication. The algorithm should be able to work under synchronous as well as asynchronous communication channels/primitives.
Existing algorithms proposed for termination detection can generally be classified according to communication patterns employed, underlying network topology, algorithm symmetry, process knowledge, communication protocol, message arrival, message optimality, and fault tolerance [12]. The most popular types of termination detection algorithms are wave, parental-relation, and credibility/recovery. A wave algorithm is one in which a message is passed to each processor in the system by a single initiator or a set of initiators. Each processor then returns information to initiator(s). These “waves” collect information about the global state of the system. Usually, wave algorithms assume an embedded logical network topology; e.g., a ring or a tree network for communicating TD (control) messages. The wave algorithms are natural solutions for termination detection. However, for parallel systems their repetitive nature and the feature that waves are following each other can be unattractive. Wave algorithms that do not have repetitive property are described in [8]. Algorithms based on parental-relations and credit/recovery are described in [5,13,15,18,19].

In algorithms based on a parental-relation, each processor is designated as child or/and parent and a tree structure is built on this relationship. A parent processor cannot be passive until all of its children are passive. When the root of the tree becomes passive, termination is detected [5]. A credit/recovery based algorithm [18] begins with a total credit of “one” in the system. Each active processor holds a portion of the credit. When a processor becomes passive, it sends the credit it holds to an active processor. When a processor holds exactly one credit and becomes passive, termination is detected.

3. RING ALGORITHMS

In this section we describe algorithms for the AP and APS models when the communication pattern used for TD corresponds to that of a token-passing logical ring network. Let processors $PE_0,...,PE_{N-1}$ be the $N$ processors forming this logical ring. Only control messages use the ring structure and the communication characteristics of work messages are determined by only the application. The ring algorithms for both models are based on wave propagation concepts and are derived from Dijkstra’s solution for distributed computing [5]. Termination detection in the AP model requires slightly more actions and may require an additional round through the ring to conclude termination. As will be discussed in Section 5., the additional overhead arising when using an AP algorithm in an APS setting is significant for systems with large number of processors.
3.1 Algorithm Ring_APS: 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. Note that the communication of token constitutes a control message, also called a TD message. A passive processor upon 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 processor in the system. The initiator sends out another clean token (if it is still passive).

Concluding that termination actually occurred requires not only the receipt of a clean token, but the knowledge that all messages sent have been received. To achieve this, the token is augmented with two fields that keep track of the number of messages sent and received. Let \( nr_{send} \) and \( nr_{rec} \) be these two entries, 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 numbers of sends and receives issued by all processor are equal.

Initially there is no token in the system. A processor becoming passive and never having received a token, becomes an initiator and generates a clean token to be passed around. If two or more processors become initiators, they will each inject a clean token into the system. We solve this multiple initiator problem by assigning priorities to processors. For example, by using processor number based on the logical ring structure as the priority. When a token reaches a passive processor that has already initiated a token, the processor of higher priority kills or purges the token generated by the processor with lower priority. If a processor has received token(s) previously and becomes passive later, it does not issue a new token. The total number of loops performed by the ring algorithm is bounded by the total number of messages exchanged among processors. The proof of correctness for this algorithm is almost folklore. However, since it gives the basic structure of correctness arguments needed, we sketch it in the following.

**Theorem 1** The Algorithm Ring_APS detects termination in the APS environment if and only if the underlying computation has terminated.

**Proof:** Assume first that the underlying computation has terminated; i.e., all processors are passive and there are no messages in transit. The last token issued by the initiator will remain clean and the number of sends and receives will be equal when the token returns to its initiator. Hence, the initiator will declare termination.

To prove the second part of the theorem, we show that if the underlying computation has not terminated, the algorithm cannot declare termination. When computation has not terminated, there is at least one active or server processor or there is at least one message in transit. When there is at least one active or server processor, the token will be marked as dirty and the initiator will not detect termination.
Assume all processors, except $PE_i$, are passive and that $PE_i$ sends a work message to a $PE_j$ and becomes passive afterwards. At this point in time, all processors are passive, but there is a message in the network. When $PE_j$ is after $i$ in terms of the token flow direction in the loop, processor $PE_i$ receives a clean token and it remains clean (as shown in Figure 3a). If $PE_j$ receives the token before the work message, $PE_j$ leaves the token clean. When the token reaches the initiator, the number of sends will be one more than the number of receives. Hence, the initiator will not declare termination. If the token reaches $PE_j$ after the work message, then $PE_j$ will make the token dirty and no termination will follow. When $PE_j$ is before $PE_i$ in the loop and the token reaches $PE_j$ before the work message, the token remains clean at $PE_j$. $PE_i$ will keep the token clean and the initiator will receive a clean token (as shown in Figure 3b). However, the number of sends will be one more than the number of receives and the initiator will not issue termination. If $PE_j$ receives work message before the token, it makes the token dirty and no termination will follow.

3.2 Algorithm Ring_Flag: Ring Algorithm for the AP Model

The algorithm described for the APS model in Section 3.1 fails to detect termination for the AP model. The algorithm described for the APS model fails to detect termination for an AP environment. More precisely, it could conclude that termination has occurred when the system has not actually terminated. The problem arises since active processors are now able to make passive processors active and such active processors can make other passive processors active. A scenario for which the APS algorithm works incorrectly for AP model is as follows. Assume that $PEs i, j,$ and $k$ are positioned such that $PE_k$ is in between $PE_i$ and $PE_j$, going clockwise on the ring from $PE_i$. Further, assume that initially all processors except $PE_j$ are passive. The initiator, $I$, will mistakenly detect termination if

(i) $PE_j$ sends a work request to $PE_i$ and becomes passive.

(ii) $PE_i$ sends a work request to $PE_k$ and continues as active.

(iii) Clean token is somewhere between $PE_i$ and $PE_k$.

(iv) $PE_k$ receives the work request, but finishes it before the token reaches.

Note that $PE_j$ will also mark the token as clean. Processor $PE_k$ will increase $nr_{recv}$ by one and processor $PE_j$ will increase $nr_{send}$ by one. The initiator will find the token as clean and $nr_{send} = nr_{recv}$ and thus will declare the termination while $PE_i$ is still active.

![Figure 4: An example of how Ring-APS fails in AP environment.](image-url)
+1 in Figure 4 represents a message sent and –1 represents a message received.

We next describe Algorithm Ring_Flag which corrects this problem and correctly detects the termination.

In the Algorithm Ring_Flag every processor uses, in addition to \( nr_{\text{send}} \) and \( nr_{\text{rec}} \), a message flag, \( m_{\text{flag}} \). 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 \( m_{\text{flag}} \) to 0 when it receives a clean or dirty token. When a clean token reaches a passive processor and that processor’s \( m_{\text{flag}} \) 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. When multiple tokens are present, a token initiated by a lower priority processor can reset \( m_{\text{flag}} \) in \( PE_i \) and later be purged. In such a case, another token issued by a higher priority processor now arriving at \( PE_i \) may find no evidence that \( PE_i \) has sent a work message which made another processor active. We solve this problem by recording the priority of the processor whose token has reset the \( m_{\text{flag}} \). When a clean token passes through a passive processor with \( m_{\text{flag}}=0 \) and the priority of the processor initiating the token is higher than the recorded priority, the token is made dirty and the priority information is updated. In some sense, the earlier resetting of \( m_{\text{flag}} \) is realized as being done by the wrong token. This check sometimes requires one additional round of the token around the ring. When a processor sends a work message and sets \( m_{\text{flag}} = 1 \), it sets the priority as being undefined.

**Theorem 2** The Algorithm Ring_Flag detects termination in the AP environment if and only if the underlying computation has terminated.

**Proof:** The argument that termination is detected once the computation has finished is as given in Theorem 1. Assume now algorithm Ring-Flag declares termination. Consider first the situation when only one token was present; i.e. the token which resets \( m_{\text{flag}} \) to 0 comes from an initiator which eventually concludes termination. The initiator processor declares termination when (i) the token is clean and (ii) \( nr_{\text{send}}=nr_{\text{rec}} \). This means that during the last loop, every processor was passive, every message sent was received, and no new work messages were generated. As a result, there is no possibility of a processor becoming active again and hence the termination has occurred. When there are more than one token present, we need to make sure that the initiator having the highest priority does receive the correct information about the work messages sent by active processors. Recall that it is possible for a token with low priority to reset \( m_{\text{flag}} \) to 0 and then get purged. Since every processor records the priority of the token initiator after its \( m_{\text{flag}} \) is reset to 0, the token from a high priority processor will be made dirty by a processor whose \( m_{\text{flag}} \) has been reset by a low priority token. This is because previously \( m_{\text{flag}} \) was reset by the wrong token and the wrong token was made dirty. Hence, an initiator receives a clean token if and only if no work messages were sent by a processor, and correct termination detection follows.

### 3.3 Algorithm Ring_Counter: Ring Algorithm for the AP Model

The Algorithm Ring_Counter is a variation of Algorithm Ring_APS for the AP environment. The algorithm does not require the book-keeping employed in Algorithm Ring_Flag (i.e., \( m_{\text{flag}} \) and priority recording). Instead, it uses tokens of size \( O(N) \), where \( N \) is the number of processors. Each processor now keeps track of the number of messages sent to every other processor by maintaining an array \( \text{counter}_{i}[j] \), of size \( N \), where element \( \text{counter}_{i}[j] \) tracks the number of messages sent to \( PE_j \). For each received message, \( PE_i \) decrements \( \text{counter}_{i}[i] \). The token is modified to include two fields: (i) clean/dirty flag and (ii) an array of size \( N \). Clean/dirty flag functions the same as in Ring_APS. If an active processor receives a clean token, it marks it as dirty. Any processor that receives a dirty token will pass it to the next processor in the ring. Whenever a passive processor receives a clean token it will combine the local array with the
array on the token by adding the corresponding elements. When the initiator receives the token, it will issue termination only if (i) the token is clean, and (ii) every element of the array is zero. The second condition makes sure that every message sent is received by its corresponding processor. If the initiator cannot detect termination, it will clear the token and send it again in the ring. The correctness of Algorithm Ring.Counter uses arguments similar to those used in the proof for Theorem 3.2.

4. TREE ALGORITHMS

Termination detection using a logical ring for communication is characterized by repeatedly sending a token through the $N$ processors. This can become inefficient in large systems, particularly when towards the end of the program execution there are only a few active processors. In this section, we describe TD algorithms using the communication pattern of a logical tree. An example tree is shown in Figure 5. Processors becoming passive send TD information either up the tree or to a designated processor close to the root. The solutions we present belong to the parent responsibility class of termination detection algorithms [12].

In the logical tree every leaf corresponds to one of the $N$ processors. The processors are combined together in groups and one of the processors within each group is made the parent of the group. These parent processors form the second level of the tree. The third level is constructed by grouping these 2nd level parents and selecting 3rd level parents. As this scheme continues, some processors are parent at multiple levels. For example, for a binary tree with two processors per group, the first level consists of all the processors at the leaves. If processors are logically numbered from zero to $N-1$, all the processors having addresses that are multiples of two may be chosen as parents at the second level.

We first describe the tree algorithm for the APS model. For the AP model we present two algorithms which are representative of two different classes of solutions.

4.1 Algorithm Tree.APS: Tree Algorithm for the APS Model

Assume that every processor knows its level, its parent, and, if it is a parent, its number of children in the logical tree. Further, each processor is assigned one processor on the control level of the tree. On the
outset, assume again that all processors are active. In Algorithm Tree_APS, an active processor becoming passive for the first time informs its immediate parent by sending a TD message (control message) containing the number of sends and receives the processor has issued so far. The processors that are also parents at the level above report to themselves. When all the processors in a group have reported to their parent PE, the parent processor 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. When each child of the root has reported to the root and the root has become passive, the root checks the number of sends and receives. If they are equal, the root declares termination. If the number of sends and number of receives at the root do not match, there are work-request messages in transit. These will make some processors go into the server state and thus the root does not issue termination.

A passive processor goes into the server state if an active processor sends it a work request. When a processor becomes passive after being in the server state, it does not send a control message to its parent, rather it reports directly to the assigned processor at 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. Observe that the control level helps minimize the time delay that is present when only a few processors are in the server state. The tree structure helps in eliminating the network congestion that may arise due to initial control messages.

The choice of the control level can have significant impact on the performance. A control level closer to the root implies that control messages traverse fewer hops, but it increases the potential for communication bottlenecks. The control level could also be set dynamically depending on the frequency (i.e., how often) and how many processors go into the server state. Section 5. discusses these issues in more detail. We conclude with the correctness of the described algorithms for the APS model.

**Theorem 3** The Algorithm Tree_APS detects termination in the APS environment if and only if the underlying computation has terminated.

**Proof:** When the underlying computation has ended, all processors are passive and there is no message in transit. The processors enter into a passive state from either active or server state. In the case of active-passive transition the processor informs its parent, while in the case of server-passive transition it informs a processor at the control level. The root receives notification and since \( nr_{rec} = nr_{send} \), the root declares termination.

Assume that the algorithm did not declare termination. There exists one active or server processor or there is a work message in transit. If there exists at least one active processor, say \( PE_i \), then \( PE_i \) will not report to its parent and the root will not hear from at least one of its children. Hence, the root will not detect termination. Next, assume there exists no active processor, but there exists a processor, say \( PE_p \), in the server state. Since there is no active processor, all work messages have been sent and the final number of work messages is reflected in the sum of the messages sent. Hence, once the root hears from all its children we have \( nr_{send} > nr_{rec} \) and the root will not declare termination.

Finally, consider work messages in transit. Assume \( PE_i \) sends a work message to processor \( PE_j \) and becomes passive afterwards. At this point in time, all processors could be passive and the root could be informed of this. However, the root would have \( nr_{send} > nr_{rec} \) and will thus not declare termination. Correctness of the algorithm follows.

### 4.2 Tree Algorithms for the AP Model

We now turn to termination detection in the AP environment. As already stated, the total number of sends and receives can no longer be used to conclude termination. The topology of the ring allowed us to detect termination by using flags. When communication in TD follows a tree patterns, the setting of flags no
longer works, since multiple control messages can move simultaneously towards the root. We describe two different tree algorithms for the AP model. The first one, **Algorithm Tree_Counter** can be viewed as an extension of the APS tree algorithm. We make use of the control level to by-pass processors and maintain for each processor an array of size $N$. The second algorithm, **Algorithm Tree_Ack**, operates on a tree without using the control level. TD messages have constant length and a dynamic parent responsibility is used to correctly determine termination.

**Algorithm Tree_Counter**
Let $Counter_i[j]$ be an array of size $N$ stored in $PE_i$, $0 \leq i \leq N-1$. 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 $Counter_j[i]$. Thus $-Counter_j[i]$ in $PE_j$ corresponds to the number of work messages received by processor $j$.

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. Note that when a $PE_i$ as a leaf node of the tree sends a TD message to its parent, entry $-Counter_i[i]$ contains the total number of receives for $PE_i$. A processor corresponding to a non-leaf node in the tree, upon receiving TD messages from its children, merges the arrays received (merging corresponds to vector addition). Once the node has received TD messages from all its children, it sends a TD message consisting of the updated $Counter$ 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 processor at the control level, upon receiving such TD message, sends it on to the root, provided that it has received TD messages from every child, and (ii) every entry on the merged $Counter$-array is zero. The correctness of Algorithm Tree_Counter uses arguments similar to those used in the proof for Theorem .

The flow for TD messages in Algorithm Tree_Counter is analogous to the flow in the APS tree algorithm. However, TD messages contain now an array of size $N$. The impact of this increase on overall performance is studied in Sections 4.3 and 5..

**Algorithm Tree_Ack**
Algorithm Tree_Ack is based on a well-known approach for termination detection [3], static and dynamic parental responsibility. However its implementation has been optimized for the APS environment. 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 acknowledgment messages and may cause delays in termination detection.

In Algorithm Tree_Ack every processor $PE_i$ uses an array $Server_i[.]$ of size $N$, where $Server_i[j]$ records the number of work messages sent from $PE_i$ to $PE_j$ which have not yet been acknowledged, $0 \leq i \leq N-1$. In addition, every processor $PE_i$ uses an entry $M_i$ to record dynamic parental relations.

When an active processor $i$ sends a work message to processor $j$, $PE_i$ increments $Server_i[j]$. If processor $j$ is active at the time it receives the work message from $PE_i$, it sends an acknowledgment back to $PE_i$. Upon receiving this acknowledgment, $PE_i$ decrements $Server_i[j]$. If, on the other hand, processor $j$ is passive at the time it receives the work message from $i$, $PE_j$ sets $M_j = i$ and then becomes active. Once $j$
becomes passive again, it sends an acknowledgment message to processor \( M_j \), the processor that made \( j \) active.

A processor becomes passive if it completed all local work, satisfied all work request sent to it, and if all the entries in array \( Server_i \) are zero.

**Theorem 6:** Algorithm Tree-ACK for the AP model detects termination if and only if the underlying computation has terminated

**Proof:** When the underlying computation has terminated, all the processors become passive and they report to their parent, hence the root hears from all of its children and declares termination.

If the underlying computation has not terminated, there exists at least one active or server processor or there is work message in transit. Assume there is one active processor, \( PE_i \). \( PE_i \) will not report to its parent, therefore the root will not hear from all of its children and will not declare termination.

If there is at least one server processor, \( PE_i \), then there is at least one active processor, \( PE_j \), which is waiting for an acknowledgment from \( PE_i \). Since \( PE_j \) will not report to the root, the root will not declare termination.

Assume there is at least one work message from \( PE_j \) sent to \( PE_i \). Since \( PE_j \) will wait for the acknowledgment from \( PE_i \), it will not report to the root, hence the root will not declare termination.

### 4.3 Complexity Comparison of the Tree Algorithms

In this section we present an analytical comparison of three TD tree algorithms: Tree_APS, Tree.Counter, and Tree_Ack. Our comparison will provide insight into where and why the use of an AP algorithm can result in poor performance when detecting termination in an APS scenario. Our analytical conclusions are supported by the experimental results.

The asynchronous nature of termination detection coupled with the difficulty in capturing congestion in a parallel system makes a general analysis difficult. We thus concentrate on situations that we judge to be representative of scenarios arising in APS applications. For the purpose of analysis, 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 \( n \) processors is active while the other \( n-1 \) processors are passive. The active processor sends work messages to the passive processors (which then become server processors). We assume that each of the passive processors receive at least one work message and that a total of \( m \) work messages are sent, \( m \geq n-1 \). In an application, several diffusing processes are likely to execute simultaneously. We do not address the interaction between multiple diffusing processes. We analyze the best and worst case situation of a single diffusion process in terms of four parameters:

- **TD_total_msgs**: the total number of control messages sent
- **TD_delay_msgs**: the number of control messages sent **after** the last processor becomes passive
- **TD_total_comp**: the total amount of additional work incurred during a diffusing process
- **TD_delay_comp**: the amount of work done by all processors to detect termination **after** the last processor becomes passive.

<table>
<thead>
<tr>
<th></th>
<th>Tree_APS</th>
<th>Tree_Counter</th>
<th>Tree_Ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_total_msgs</td>
<td>( h+n+2c )</td>
<td>( h+n+2c )</td>
<td>( m+h )</td>
</tr>
<tr>
<td>TD_delay_msgs</td>
<td>( c )</td>
<td>( c )</td>
<td>( h )</td>
</tr>
<tr>
<td>TD_total_comp</td>
<td>( O(n) )</td>
<td>( O(nN) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>TD_delay_comp</td>
<td>( O(c) )</td>
<td>( O(cN) )</td>
<td>( O(h) )</td>
</tr>
</tbody>
</table>

Table 1: Best case comparison for a single diffusion process
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 (in algorithms making use of a control level). Note that \( c < h \) and \( 2^c << N \).

We use the following diffusing process scenario to capture best case performance: (i) passive processors become servers exactly once (i.e., if a processor receives more than one work message, work messages arrive while the processor is in the server state) and (ii) the times at which the \( n-1 \) server processors become passive minimize the four quantities stated above. Table 1 summarizes the bounds for the three tree algorithms under these assumptions. The total number of control messages, \( TD_{\text{total_msgs}} \), given for Tree_APS and Tree_Counter is obtained by including \( h \) control messages sent from the active processor up the tree (after becoming passive), one control message for each of the \( n-1 \) processors to a processor on the control level of the tree (these control messages are sent after switching from server to passive), and \( 2^c \) control messages for handing the termination detection process from level \( c \) of the tree to the root. After the last processor becomes passive, \( TD_{\text{delay_msgs}} \), is bounded by the length of the path from this processor to the root (it is either \( c \) or \( h \)). The total computation, \( TD_{\text{total_comp}} \), differs depending on the token size (\( O(1) \) for Tree_APS and Tree_Ack, \( O(N) \) for Tree_Counter) and on whether every work message results in a control message. For the best case scenario, \( TD_{\text{delay_comp}} \) is determined by \( TD_{\text{delay_msgs}} \) and the token size.

<table>
<thead>
<tr>
<th></th>
<th>Tree_APS</th>
<th>Tree_Counter</th>
<th>Tree_Ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( TD_{\text{total_msgs}} )</td>
<td>( h+m+c+m )</td>
<td>( h+m+c+m )</td>
<td>( m+h )</td>
</tr>
<tr>
<td>( TD_{\text{delay_msgs}} )</td>
<td>( n+2^c )</td>
<td>( n+2^c )</td>
<td>( m+h )</td>
</tr>
<tr>
<td>( TD_{\text{total_comp}} )</td>
<td>( O(m) )</td>
<td>( O(mN) )</td>
<td>( O(m+h) )</td>
</tr>
<tr>
<td>( TD_{\text{delay_comp}} )</td>
<td>( O(n) )</td>
<td>( O(nN) )</td>
<td>( O(m+h) )</td>
</tr>
</tbody>
</table>

Table 2: Worst case comparison for a single diffusion process

For a worst case scenario of a diffusing process we assume that (i) if a processor receives \( k \) work messages, it switches from passive to server state \( k \) times and (ii) every new passive state results in a maximum number of new control messages. Table 2 summarize the bounds for the three algorithms for this worst-case scenario. For algorithms Tree_APS and Tree_Counter this worst case scenario results in a considerable increase in the total number of control messages. The quantities shown in the table captures the case when the initially active processor generates \( h \) control messages, each one of the \( m \) work requests generates one control message (from the processor issuing the work message), and a total of \( cm \) control messages is generated by the \( n-1 \) server processors.

In summary, algorithms Tree_APS and Tree_Counter send the same number of control messages. They differ in the size of the control messages (\( O(1) \) versus \( O(N) \)) and thus the computation required (\( O(n) \) versus \( O(nN) \)). 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. This dependence between work messages and control messages is clearly reflected in the experimental results.

5. 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. Due to the asynchronous nature of the termination detection operation, it is difficult to analyze its impact on the underlying application by using complexity analysis only. In the following we also experimentally evaluate the performance of the proposed algorithms. Our study uniquely focuses on two measures for evaluating termination detection algorithms:
termination delay: corresponds to the time between the last processor becoming idle and the root or initiator detecting termination. The termination delay measures characteristics of the underlying TD algorithm in the final stages.

slowdown: measures the impact of a termination detection algorithm on the overall execution time. The slowdown is the result of the TD algorithm and the TD messages it generates. Let $T_I$ be the time needed by an application in an idealized situation in which termination detection is “free” and let $T_D$ be the cost when termination detection is performed. The slowdown can be viewed as $T_I/T_D$.

The experimental results do not compare achieved slowdown against the ideal situation, but rather compare it in terms of the total execution time and the number of TD messages of different TD algorithms. Observe that it is possible to have a TD algorithm with short termination delay, but large slowdown.

Our experimental work demonstrates 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 performance loss is greater than what we expected after carefully minimizing the number of control messages for the AP algorithms. We also provide experimental results and demonstrate the efficiency of the APS algorithms.

The Cray T3E used in our experiments consists of 272 high-performance Digital Alpha 64-bit microprocessors, running at 300 MHz. The T3E’s topology is that of a three-dimensional torus. Each processor runs UNICOS/mk, a UNIX operating system. The memory is logically shared and physically distributed, with each PE having 128 MB (16 MW). The performance results have been obtained over multiple runs and are averaged over three best runs.

5.1 Application Simulation Environment

While the performance of a termination detection algorithm is independent of the computations employed by the underlying application, it is heavily dependent on the disparity of the workload across processors and on the amount and pattern of communication exhibited by the application. In order to carry out the performance evaluation of the proposed TD algorithms under different application scenarios, we have implemented a comprehensive application simulation testbed that allows us to vary the workload at each processor and the amount as well as the pattern of communication embedded in an application. The testbed is implemented using statistical processes based on Poisson distribution model for workload at each processor. We have used Poisson mean values of 1, 2, 5, 20 and 40 where mean value of 1 implies that initially almost all PEs get similar workloads and thus finish very close to each other. The higher mean values imply increasing disparity among the initial workloads at different processors. Within each processor, the workload consists of local and remote work and different percentages of local and remote work have been used. Based on this application testbed we classify the application scenarios used in our experiments into different categories as specified below:

- **Class A Applications:** Workload distribution across processors is almost balanced. This scenario characterizes all the processors arriving at the termination point almost simultaneously.
  - Class A1: low communication activity among processors
  - Class A2: medium communication activity among processors
  - Class A3: heavy communication activity among processors

- **Class B Applications:** Workload distribution across processors is imbalanced. This scenario characterizes different groups of processors arriving at the termination point at different points in time.

3 Results reported in this paper are for machine size of 128 PEs, unless otherwise stated.
- Class B1: low communication activity among processor
- Class B2: medium communication activity among processor
- Class B3: heavy communication activity among processors

- **Class C Applications**: Workload distribution across processors is highly imbalanced.
  - Class C1: low communication activity among processor
  - Class C2: medium communication activity among processor
  - Class C3: heavy communication activity among processors

The purpose of changing initial workloads is to test the algorithms under conditions where PEs become idle at different times and different communication volume result in different processor interactions. Pseudo code of the main application loop used in the simulations is shown below.

```plaintext
// workload     : Initial workload
// mean         : Poisson mean for generating initial workloads
// prob_work_gen: Probability of generating work requests
// N            : Number of processors
// float rand() : returns a random number between 0 and 1
// int trunc()   : truncates a number to the closest integer
// send(int d, data) : send to destination d
// recv(int s, data) : receive from source s

workload = POISSON(mean) * constant;

// Main loop body
while (workload--) {
    if (ran() < prob_work_gen) { // Generate work request
        dest = trunc(rand() * N); // Destination processor
        if (ran() < 0.5) {
            // Unacknowledged work request
            send(dest, some_work);
        } else {
            // Acknowledged work request
            send(dest, some_work);
            recv(dest, acknowledgement); // Wait for acknowledgement
        }
    }
}
```

Main Loop Body used in the implementations to generate workload in each processor.

The rest of the section is organized as follows. We first compare different Ring algorithms in Section 5.1 and study their performance in terms of TD delay and slowdown metrics. The best Ring algorithm is then compared with Tree algorithms in Section 5.3 and further experimental analysis of different Tree algorithms is carried out assuming different application scenarios. Finally, the best Tree algorithm is further evaluated based on different properties of the tree structure employed.

### 5.2 Performance Evaluation of Ring Algorithms

Figure 6(a) compares the termination delay for three ring algorithms on various machine sizes. These algorithms include the three ring algorithms described in Section 3. We refer to them as algorithms Ring_APS, Ring_FLAG and Ring_COUNTER. The termination delay in Figure 6 corresponds to the average termination delay over all classes of applications. 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 for this is Algorithm Ring_FLAG has the potential need to make one additional through the ring and Algorithm Ring_COUNTER suffers due to large token size which is proportional to the number of processors.

Figure 6: Termination detection delay for three ring algorithms in machines of different sizes, results averaged over all classes of applications.

Figure 7: Slowdown in terms of total execution time caused by different ring algorithms for different Class B applications. Note that in class B applications there is a moderate imbalance in workload among processors.

Figure 7 shows the slowdown effect in terms of total execution time of the ring algorithms for different Class B applications. As the amount of communication increases, one expects an increase in the total execution time. However, it increases differently for each algorithm. The figure clearly shows that for larger machine sizes (128, in this case), the use of an AP algorithm for an APS application results in a significant loss in performance. The use of counters and thus TD messages of size \( O(N) \) contributes to the large total execution time. Algorithm Ring_FLAG gives a reasonable performance for small remote work loads, but it deteriorates like Ring_COUNTER for higher remote work loads. We conclude that Algorithm Ring_APS performs significantly better both in terms of termination delay and slowdown measures for the applications that employ APS characteristics.
5.3 Performance Evaluation of Tree Algorithms

First of all we compare the performance of the best Ring APS algorithm with the tree algorithms. Figure 8 compares the termination detection delay of Ring_APS and tree based algorithms for different machine sizes bigger than 31 processors. We have observed that for machine sizes smaller than 32 processors, the advantage of a tree algorithm diminishes rapidly due to the overhead involved in maintaining a tree structure. 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.

Although the performance of the tree algorithms in Figure 8 appears very similar, Figure 9 gives a closer look at the performance of the tree-based algorithms. The performance curves correspond to the termination delay averaged over all application classes. In terms of termination detection delay, the performance of the Tree_APS and the Tree_ACK is not strikingly different. On the other hand, the effect of large token is obvious on Algorithm Tree_COUNTER.

![Figure 8: Comparing Ring_APS and two tree algorithms assuming different machines sizes, results averaged over all classes of applications.](image)

![Figure 9: Termination detection delay, comparing tree algorithms; results averaged over all classes of applications.](image)
Figure 10: Slowdown in terms of total number of TD messages induced by different tree algorithms in a 64 processor machine running Class B applications where processors have medium imbalance among their workloads.

Figure 11: Average number of TD messages induced by different tree algorithms assuming different classes of applications on a 64-processor machine.

Now we take a closer look at the tree algorithms in terms of slowdown. As shown in Figure 10, capturing slowdown in terms of total TD messages, the Tree_ACK induces twice the number of TD messages into the network compared to Tree_APS and Tree_COUNTER algorithms for different class B applications. We also evaluate the performance across Classes A, B, and C (shown in Figure 11). In all the cases, the number of TD messages generated by algorithm Tree_COUNTER is close to the number of TD messages generated by Tree_APS. This is expected, because the two algorithms only differ in terms of the
TD token size. The algorithm Tree_APS uses a $O(1)$ size token whereas the Tree_COUNTER uses $O(N)$ size token. The communication structure is exactly the same in both the algorithms. However, when slowdown is compared in terms of total execution time, the results are very different. The algorithm Tree_COUNTER slows down the underlying computation significantly, as shown in Figure 12. This also results in a higher termination detection delay as shown in Figure 9. Note that Tree_APS algorithm outperforms Tree_ACK and Tree_COUNTER algorithms in APS environment, both in terms of termination detection delay and overall computation slowdown.

![Figure 12: Slowdown in terms of total execution time induced by different tree algorithms assuming different application scenarios of Class A applications. Recall that in Class A applications processor reach termination point very close to each other.](image)

In the next few graphs we evaluate the performance of the tree structure of Algorithm Tree_APS in terms of the control level and the node degree. Recall that when a server processor becomes passive, it sends a control message to a processor on the control level. When the control level is closer to the root, it takes fewer messages to deliver information to the root. However, a control level closer to the root can lead to congestion.

![Figure 13: The impact of control level on termination delay in Algorithm Tree_APS.](image)
Figure 13 demonstrates the effect of different control levels for algorithm Tree_APS for all classes of applications. When the control level is 1, the parents of the immediate leaves represent the control level and one can conclude that it is almost like having no control level at all. By placing the control level higher up in the tree, performance improves for all considered workloads. The improvement is significant for Class A applications because most of the processors arrive at the termination point very close to each other thus causing potential congestion. Moving control level up the tree helps distribute this congestion over multiple levels. On the other hand, in Class C applications, due to the high disparity of workload among processors, processors arrive at termination at different points in time. In this case, there are few control messages sent at the same time and performance is almost independent of the position of the control level.

The degree of the nodes in a TD tree represents the group size and it is also a parameter of the tree-based algorithms. Whether large or small group sizes give a better performance depends on the initial workloads and the amount of communication employed. Figure 14 shows a typical behavior. Our results show that (i) too small group sizes are not ideal due to the associated larger tree height and (ii) large group sizes are not ideal due to the arising congestion. The group size should be chosen so that a compromise between tree height and congestion is achieved. For example, for a 128-processor T3E, a group size of 16 gives the best performance for all load types used.

![Figure 14: The impact of node degree on the total execution time in Algorithm Tree_APS.](image-url)

6. CONCLUSION

We considered two models for termination detection arising in data-driven applications, the APS model and the AP model. For each model we described a number of different algorithms that adapt and modify methods for distributed termination detection algorithms to parallel systems. The algorithms differ on whether the communication between control messages occurs on a logical ring or tree structure, the size of the control messages, and the type of acknowledgment messages needed. In the experimental work we measured termination delay and slowdown. Our results show that under almost all circumstances the use of an AP algorithm for an APS scenario results in a considerable loss of performance. Our conclusion is that (i) AP algorithms are unable to make effective use of the more restricted communication patterns arising in APS applications and (ii) APS applications should avoid using general termination detection implementations.
References


