An Efficient Delay-Optimal Distributed Termination Detection Algorithm*

Nihar R. Mahapatra
mahapatr@cse.buffalo.edu
Dept. of Computer Sc. & Eng.
Univ. at Buffalo — The State Univ. of New York
Buffalo, NY 14260-2000

Shantanu Dutt
dutt@eecs.uic.edu
Dept. of Electrical & Computer Eng.
University of Illinois at Chicago
Chicago, IL 60607-7053

Abstract

One of the important issues to be addressed when solving problems on parallel machines or distributed systems is that of efficient termination detection. Numerous schemes with different performance characteristics have been proposed in the past for this purpose. These schemes, while being efficient with regard to one performance metric, prove to be inefficient in terms of other metrics. A significant drawback shared by all previous methods is that they may take as long as $\Theta(P)$ time to detect and signal termination after its actual occurrence, where P is the total number of processing elements. Detection delay is arguably the most important metric to optimize, since it is directly related to the amount of idling of computing resources and to the delay in the utilization of results of the underlying computation. In this paper, we present a novel termination detection algorithm that is simultaneously optimal or near-optimal with respect to all relevant performance measures on any topology. In particular, our algorithm has a best-case detection delay of $\Theta(1)$ and a finite optimal worst-case detection delay on any topology equal in order terms to the time for an optimal one-to-all broadcast on that topology—we derive a general expression for an optimal one-to-all broadcast on an arbitrary topology, which is an interesting new result in itself. On k-ary n-cube tori and meshes, the worst-case delay is $\Theta(D)$, where D is the diameter of the architecture. Further, our algorithm has message and computational complexities of $O(\max(MD, P))$ ($\Theta(\max(M, P))$ on the average for most applications—the same as other message-efficient algorithms) and an optimal space complexity of $\Theta(P)$, where M is the total number of messages used by the underlying computation. We also give a scheme using counters that greatly reduces the constant associated with the average message and computational complexities, but does not suffer from the counter-overflow problems of other schemes. Finally, unlike some previous schemes, our algorithm does not rely on FIFO ordering for message communication to work correctly.

Keywords: Broadcast, detection delay, distributed computation, k-ary n-cubes, message complexity, message passing, termination detection.

*S. Dutt was supported by NSF grant MIP-9210049.

1 Introduction

1.1 Motivation and Background

In this paper, we consider efficient algorithms for detecting termination of parallel and distributed computations. The model of the computing system used is the same as that in previous work [2] and basically consists of an asynchronous network of P processing elements (PEs) labeled $0, 1, \ldots, P-1$ with diameter D. Each PE is connected to one or more PEs, known as its *neighbors*, by bidirectional links. A PE may send messages to or receive messages from its neighbors along the bidirectional links connecting them. Although message passing between only neighboring PEs is common in parallel and distributed algorithms, there are many parallel algorithms which require message passing between PEs at arbitrary distances from each other [6]. The only effect that the message-passing behavior of the parallel/distributed algorithm has on a termination detection algorithm is that the detection delay (to be defined shortly) of the termination detection algorithm may be more in the case where arbitrary PEs communicate compared to the case in which only neighboring PEs communicate. In this paper, our main discussion will pertain to the neighbor-neighbor message-passing case, but we will point out any significant differences in the other case. The parallel/distributed computation whose termination is to be detected is termed the *primary computation* and messages used by it are called *primary messages*. The total number of primary messages is denoted by M. The computation associated with the termination detection algorithm is called the *secondary computation* and the messages used by it are termed *secondary messages*.

The following features characterize the primary computation: (1) At any time, a PE can be either *busy*, or otherwise *idle*, as follows.

Definition 1 A PE is *busy* if it has some primary computation to perform, otherwise it is idle.

(2) Only a busy PE may send primary messages to its neighbors via its adjacent communication

links. (3) A busy PE becomes idle after completing the part of the primary computation assigned to it. (4) PEs can receive primary messages in both busy and idle states. (5) An idle PE becomes busy when and only when it receives a primary message. Thus an idle PE cannot spontaneously become busy (except possibly when the primary computation begins). We assume that initially all PEs are assigned some primary computation, i.e., initially all PEs are busy (generalization of the discussions in this paper to the case when only some of the PEs are initially busy is straightforward). The termination detection problem lies in one (or all) PEs inferring the completion of the primary computation. Clearly, for the primary computation to be complete not only must all PEs become idle, but also there should be no primary messages in transit since idle processors receiving such messages can become busy. Therefore termination detection algorithms need to ensure both these conditions are simultaneously met before signaling termination.

To analyze the performance of termination detection algorithms we will use the following four metrics: (1) *Detection delay* T_d defined as the time between the actual completion of the primary computation and its subsequent detection; the detection delay of termination detection algorithms when message passing is between arbitrary PEs will be denoted by T'_d; (2) *Message complexity* M_s which is the total number of secondary messages used over all PEs; (3) *Space complexity* S_s measured by the total memory used over all PEs by the secondary computation; and (4) *Computational complexity* C_s over all PEs of the secondary computation. Since information about the states of all PEs is essential for termination detection, and since $\Theta(D)$ is the minimum time in which this information can be obtained, the worst-case detection delay T_d (or T'_d) is at least $\Theta(D)$. It is shown in [5] and [2] that a worst-case lower bound on the message complexity M_s of any termination detection algorithm is $\Theta(\max(M, P))$. Since each secondary message is associated with some computational overhead, the worst-case computational complexity C_s of any termination detection algorithm also has the same lower bound. Finally, each PE needs to store information such as its label and state (busy or idle) information, so that the space complexity S_s is at least $\Theta(P)$. Therefore we obtain:

Theorem 1 *Any termination detection algorithm has, in the worst case, a detection delay of* $\Omega(D)$*, message complexity of* $\Omega(\max(M, P))$*, space complexity of* $\Omega(P)$*, and computational complexity of* $\Omega(\max(M, P))$*, where* M *is the number of primary messages used by the primary computation and* D *is the diameter of the target topology.*

1.2 Previous Work and New Results

A number of termination detection algorithms with different performance characteristics have been proposed in previous work [2, 10, 11, 12, 14, 16]. Some of these algorithms are efficient in terms of the number of secondary messages used [2, 14], but may take a long time to detect termination [2] or incur significant computational overhead [14]. Similarly, some of the algorithms are either less compute intensive [12] or require less memory [11], but are vulnerable to underflow [12] or overflow [11] problems. Recall that accounting for in-transit primary messages is crucial to correct termination detection. This is most commonly achieved by either one of two approaches: by using acknowledgements for primary messages or by counting the total number of primary messages sent and received by PEs. In Sec. 4, we will compare the performance of our algorithm with three of the most efficient algorithms proposed previously—two acknowledgement-based algorithms [2, 14] and one message-counting-based algorithm [11]; we briefly describe and analyze these algorithms next.

Before we consider any algorithms, it would be useful to briefly discuss a general technique that [2] presents that allows a termination detection algorithm to work correctly even when started at an arbitrary time after the primary computation has commenced. For this to be possible, we need to account for in-transit primary messages just before the termination detection algorithm begins, since the termination detection algorithm accounts for all primary messages sent out after it begins. It is assumed that message communication between PEs is in first-in first-out (FIFO) order, i.e., messages sent out by any PE i to another PE j are processed by the latter in the same order as they were issued by i. Before the termination detection algorithm is initiated, a "warning message" is sent out on every communication link in a certain order to flush out any in-transit primary messages. Then all primary messages sent out after the "warning message" are accounted for by the termination detection algorithm.

The flush-out operation requires as many messages as the number of communication links $|E|$. The advantage of being able to start the termination detection algorithm at an arbitrary time is that its complexity then depends not on the total number of primary messages M, but rather on the number of primary messages $M' \leq M$ sent out after the termination detection algorithm has begun. Therefore in an application in which M is likely to be more than $|E|$ and in an architecture that has the FIFO communication feature, the flush-out option will prove useful. However, exercising the flush-out option introduces nondeterminism into the detection delay since it will then depend upon whether the flush-out operation completes before or after the primary computation terminates. If the flush-out operation completes (and hence the termination detection algorithm starts) before the primary computation terminates, then the detection delay is the actual time t_1 taken by the termination detection algorithm to detect termination. For most efficient termination detection algorithms, t_1 is deterministic, e.g., $O(D)$, $O(P)$. On the other hand, if the flush-out operation completes t_2 time after the primary computation terminates, then the detection delay is $t_1 + t_2$. Here t_2 is nondeterministic, since, even though the time to execute the flush-out operation is deterministic, the time when the primary computation terminates is not known, thus making the detection delay nondeterministic. In order to keep the detection delay deterministic so that different termination detection algorithms can be compared, and to avoid the restrictive assumption of FIFO capability, we assume in all our performance analyses that the flush-out procedure is not used. Therefore we use M instead of M' in all complexity expressions. Moreover, the flush-out procedure is applicable to all termination detection algorithms considered in this paper; therefore the above assumption is a fair one as far as performance comparison of the different algorithms is concerned.

1.2.1 Acknowledgement-Based Algorithms

The algorithm in [2] embeds a spanning tree in the target topology for termination detection and uses acknowledgement messages to keep track of in-transit primary messages. At any time, a PE may be *loaded*, or *free*, as defined next.

Definition 2 A PE is *free* if and only if it is idle and all primary messages sent by it to other PEs have been acknowledged, otherwise it is *loaded.*

Initially, all PEs are assumed loaded with primary computation by their parents in the tree. A primary message issued by a PE is acknowledged only after the primary computation associated with that message is completed at the recipient and other PEs that may subsequently receive primary messages from the recipient PE. Thus if PE i sends a primary message to PE j which subsequently sends a primary message to PE k, then PE j sends an acknowledgement to PE i when it becomes idle and has received acknowledgements for all primary messages (e.g., the one sent to k) issued by it after receiving the primary message from PE i. Consequently, if there is a sequence of (not necessarily distinct) PEs $(p_1, p_2, \ldots, p_m)$ such that p_i receives a primary message from p_{i-1} and subsequently sends a primary message to p_{i+1}, where $2 \leq i \leq m-1$, then p_i will send an acknowledgement to p_{i-1} only after it receives one from p_{i+1} and becomes idle. A PE reports a STOP to its parent in the spanning tree when it becomes free and, in case it is a non-leaf PE, has also received STOPs from all its child PEs. The root PE signals termination after it becomes free and has received STOPs from all its children. This algorithm always uses $\Theta(M)$ secondary messages, i.e., the acknowledgement messages, which is worst-case optimal. However, since the M primary messages can form a sequence, as mentioned earlier, of $\Theta(M)$ PEs, T_d in the worst case is $\Theta(\max(M, D))$, which is quite high, since generally $M = \Omega(P)$. If message passing is between arbitrary PEs, then in the worst case, we may have a sequence of $\Theta(M)$ PEs in which consecutive PEs are $\Theta(D)$ distance apart, so that T'_d in the worst case is $\Theta(MD)$. The worst-case space complexity is $\Theta(M)$ to store labels of the $\Theta(M)$ PEs to which acknowledgements need to be sent, and the computational complexity is $\Theta(\max(M, P))$.

The algorithm of [14] is similar to that of [2] described above except in the following respects. In [14], the set of loaded PEs at any time belong to a rooted tree such that all PEs, except the root PE, are considered loaded by their parents. When a free PE i receives a primary message m from a loaded PE j, it is considered to be loaded by j and becomes a part of this tree with j as its parent, thus changing the structure of the tree. As in [2], PE i acknowledges receipt of message m from j only after it becomes free. However, unlike that in [2], all messages received by i when it is loaded are acknowledged as soon as it becomes idle. Such messages do not affect the structure of the tree of loaded PEs in [14]. Again as in [2], STOP messages are sent up this tree of loaded PEs starting at leaf PEs as they become free. Note that although the tree of loaded PEs begins as a particular spanning tree with some root PE, the tree structure is dynamic throughout the primary computation except for the initial root PE which always remains the root. If we denote the final tree of loaded PEs when the primary computation terminates by $\mathcal{T}_f$, then, ignoring the time PEs take to process STOP messages, termination will be detected in H_f time units, where H_f is the height of $\mathcal{T}_f$. In the worst case, $\mathcal{T}_f$ will be a tree with the maximum height H_{max} that can be embedded in the target architecture, so that the worst-case T_d is $\Theta(H_{max})$; in most topologies, a Hamiltonian path can be embedded, so that frequently $H_{max} = P$. When message passing is between arbitrary PEs, then in the worst case there can be one or more root-to-leaf paths in $\mathcal{T}_f$ such that consecutive PEs on such paths are $\Theta(D)$ distance apart. Therefore, ignoring link contention on different root-to-leaf paths with common links, T'_d in the worst case is $\Theta(H_{max}D)$. A useful feature of the algorithm in [14] is that to reduce the number of secondary messages used on the average, the algorithm uses an array counter in each PE to keep track of primary messages received from different PEs. Then a single acknowledgement (along with the number of primary messages received) is used to acknowledge those primary messages.

1.2.2 Message-Counting Based Algorithm

Finally, we consider the algorithm in [11] which uses a message-counting technique. It uses a marker that traverses a cycle of length C (i.e., there are C PEs on this cycle) embedded in the network spanning all PEs—since this cycle may not necessarily be simple, $C = \Omega(P)$. Each PE i stores the number of primary messages sent and received by it in variables $sntp_i$ and $recp_i$, respectively. The marker has variables $sntm$ and $recm$ which store the total number of primary messages sent and received in the network. Initially, all variables are set to zero. When the marker visits a PE i, variables are updated by PE i when it becomes idle as: $sntm := sntm + sntp_i$, $sntp_i = 0$, $recm := recm + recp_i$, and $recp_i = 0$. Then the marker is passed on to the next PE in the cycle. When in a sequence of C visits, the marker continuously finds that $recp_i = 0$ at the start of each visit to the corresponding PE i, and $sntm = recm$ at the end of the last visit in the sequence, it signals termination. This algorithm can have counter-overflow problems. The detection delay of this algorithm remains the same irrespective of whether message passing is between only neighboring or between arbitrary PEs and is on the same order as the number of links in the above cycle. It can be easily shown that a cycle containing all PEs can be found in any architecture such that the number of links on the cycle is $\Theta(P)$,[1] so that $T_d = \Theta(P)$. The performance of the above three algorithms are summarized in Table 1.

[1]Such a cycle can be formed, for instance, as follows. Find a tree in the (connected) *PE graph* corresponding to the target architecture by a breadth-first or depth-first search in the PE graph—vertices and edges in the PE graph correspond to PEs and links in the architecture. Then the traversal path (that includes backtracks) corresponding to a depth-first search in this tree from some root PE is a cycle since the search starts at the root PE, visits all other PEs, and then finishes at the root PE. Moreover, since each edge in the tree is traversed exactly twice, and the tree has $P - 1$ edges, the cycle has $2(P - 1)$ links.

1.2.3 Shortcomings of Previous Work and New Results

Previous termination detection algorithms meant for general network topologies, e.g., [2, 14, 11], while being efficient with respect to one performance metric, turn out to be inefficient with regard to other metrics. One significant drawback common to all such general-purpose schemes known to the authors is that their worst-case detection delay is $\Omega(P)$. This is quite high for most networks of reasonable connectivity compared to the optimal complexity of $\Theta(D)$. Note that the cube-connected cycles and mesh architectures (for instance, that of the Intel Paragon) have degrees of only three and four, respectively, but their diameter is only $2.5 \log P$ and $\sqrt{P}$, respectively [1]. Of the four metrics discussed in Sec. 1.1, detection delay is arguably the most important metric to optimize. Since for most applications the computation and space complexities of the termination detection algorithm is modest compared to that of the primary computation, these two complexities are the least important. The message complexity of termination detection algorithms is important, especially since in the worst case, secondary messages on the order of the number of primary messages are required. However, secondary messages are usually very short length messages and hence for most applications they will consume a much smaller fraction of the available communication bandwidth compared to primary messages.

There are several factors that make detection delay the most important metric to optimize. First, detection delay signifies a waste of computing resources of the parallel or distributed system. This is accentuated by the fact that the gap between processing speed and communication speed is widening every year [18, 20], and since detection delay is primarily dependent upon inter-PE communication delays, for the same detection delay, more computing cycles are wasted in newer systems. The situation is especially severe in a distributed system of, say, workstation clusters connected by a local area network (LAN), since in this case communication times are even longer. Second, in many applications, results from a primary computation cannot be reliably used before ascertaining its termination, leading to undue delay in their utilization. Third, sometimes an application consists of many phases, where a new phase can begin only after termination of the previous phase has been detected [3]. This requires the use of multiple termination detection algorithms one after the other and aggravates both of the above problems. In Sec. 2, we present a new termination detection algorithm that has a best-case detection delay of $\Theta(1)$ and a finite optimal worst-case detection delay on any topology equal in order terms to the time for an optimal one-to-all broadcast on that topology—on k-ary n-cube tori and meshes, the worst-case delay is $\Theta(D)$. Thus it minimizes the above problems. Furthermore, in Sec. 4 we show that our algorithm has optimal or near-optimal values for other performance metrics as well.

2 New Termination Detection Algorithm

2.1 Basic Idea

We now present our new termination detection algorithm which, like the algorithms in [2, 14], uses a spanning tree for termination detection. This spanning tree, called the *termination tree*, is rooted at a "root" PE and is chosen so as to optimize a *one-to-all broadcast* from the root PE.[2] By definition, termination is reached when the primary computation is complete. In a parallel primary computation, at any time the primary-computation load is either with PEs or is extraneously present in primary messages that will finally reach a destination PE. In our algorithm, at all times, we regard (irrespective of whether it is actually the case or not) the entire primary computation as "originating" at the root PE and then "branching out" from there to all PEs via "transfers" from a PE to its child PEs in the termination tree. In natural fashion then, termination is detected at the root PE by an inverse process of "branching in," in which starting at leaf PEs, child PEs notify their

[2]This can most often be done by choosing a spanning tree of minimum depth with the "root" PE being a center of the spanning tree [19]. The *center* of a tree is defined as a vertex with the minimum distance to the furthest vertex from it; any tree has at most two centers [7].

parent PE when they have finished their primary computation. Since the longest root-leaf path in the tree is $\Theta(D)$ long, the branching-in process, and hence the detection delay of our algorithm, is related to this $\Theta(D)$ path length.

The key to our algorithm's reduced detection delay is that it is structured so that we are consistently able to regard the primary computation at any PE as originating from its parent in the termination tree, and hence by extension from the root PE. Although the algorithms in [2, 14] also start with a spanning tree and initially have the same perspective with regard to primary-computation loads at PEs, they do not maintain that perspective. In [2] (see Sec. 1.2.1), a PE at any time is considered to have received its primary-computation load from all PEs that have sent it a primary message which it has not yet acknowledged. This therefore leads to a worst-case chain of $\Theta(M)$ PEs in which a PE needs to send an acknowledgement to the preceding PE and it can only do so after receiving one from the next PE in the chain. In [14], a PE i is considered to have received its primary computation from a PE j that sent the most recent primary message to it when it was free. Again, we see that the PE which is regarded as the source of the primary-computation load at a given PE changes depending upon how primary messages are transmitted. In the worst case, we may have a sequence of $\Theta(P)$ PEs in which each PE is regarded as the origin of the primary-computation load at the following PE. This necessitates a sequence of $\Theta(P)$ acknowledge messages or a $\Theta(P)$ detection delay. Below we describe our algorithm and explain how we maintain the consistent perspective mentioned above to reduce detection delay while guaranteeing correctness.

2.2 Algorithm Description

In our algorithm, at any time, a PE can be either *busy* or else *idle*, *loaded* or else *free*, and *active* or else *inactive*—the first four terms have been defined in Defs. 1 and 2; the last two will be defined a little later. Note from Defs. 1 and 2 that if a PE is busy, it must be loaded, and if it is free, it must be idle. So the only combination of these states that a PE can have are (busy, loaded), (idle, loaded) and (idle, free). Consider first the simpler case of termination detection in which no primary messages are used. In this case, a non-root PE i reports a STOP message to its parent once it is free (which is the same as i being idle since no primary messages are used) and has received STOP messages from all its child PEs, if any. By doing so, i essentially notifies its parent that all primary-computation load that had branched out from i has been processed. Thus STOP messages are passed up the tree starting at free leaf PEs until the root PE receives STOPs from all its children. Finally, when the root PE also becomes free, it means that all primary computation is complete, and hence the root PE signals termination by broadcasting a TERMINATION message to all PEs.

Next, consider the more general case in which primary messages may be used. In this case, we use two extra messages ACKNOWLEDGE and RESUME so that we can view as before the primary-computation load at any non-root PE as originating from that PE's parent, and by extension from the root PE. A primary message $M_{i,j}$ originating at PE i and destined for a neighbor PE j is said to be "owned" by i until an ACKNOWLEDGE is received for that message. If PE j has not yet reported a STOP to its parent, then we view the primary-computation load associated with $M_{i,j}$ as being part of the existing primary-computation load at j. In this case, recipient PE j sends an ACKNOWLEDGE message to sender PE i right away.

However, if PE j has already reported a STOP to its parent before receiving $M_{i,j}$, it "resumes" sending a RESUME message upward in the termination tree. The RESUME message is sent to nullify a STOP message previously transmitted along this path from j. Note that at this time the root PE will not have signaled termination, since PE i, the sender of $M_{i,j}$, has not yet reported a STOP. The RESUME message from j travels upward until it encounters an ancestor PE k that has not reported a STOP message to its parent (either because it is not free or because it has not received STOP messages from all its children in the termination tree). The nullification of the STOP messages by RESUME messages means that subsequently any PE on the path from PE j to k can report a STOP to its parent only after it receives one from its child on the path. Thus the primary-computation load of $M_{i,j}$ now with PE j can again be viewed as originating at the root PE and having come to PE j via a sequence of transfers along the path from the root PE to k to j in the termination tree. The ancestor PE k receiving the last RESUME message then sends an ACKNOWLEDGE for the message $M_{i,j}$ down the termination tree to PE j from where it is passed onto the neighboring sender PE i. On receiving the ACKNOWLEDGE message, i "relinquishes" ownership of $M_{i,j}$ originally sent to j and can report a STOP whenever it becomes free and has received STOPs from all its child PEs. When message passing is between arbitrary PEs, the ACKNOWLEDGE message from ancestor k is directly sent to sender PE i.

Note that when primary messages are used, a PE may become alternately busy and idle, and loaded and free. Moreover, a PE may also report STOP and RESUME messages alternately. We will refer to PEs as active or inactive as follows.

Definition 3 A PE is *active* if it has either not sent any STOP messages to its parent, or if it has not sent a STOP after the last RESUME, otherwise, it is *inactive.* The root PE is *active* until it signals termination, after which it becomes *inactive.*

From the definition it follows that if a PE is inactive, it must be free, and hence idle. Moreover, a free PE will be active if it has either not received STOPs from all its children, or if it has not received STOPs from all its children after sending out the last RESUME. Initially, all PEs start out as active, and subsequently PEs may become alternately inactive and active a number of times (along with alternately idle and busy, and free and loaded, as stated previously). A formal description of the above termination detection algorithm called DETECT_TERMINATION is given in Fig. 1.

Algorithm DETECT_TERMINATION(i)
/* Algorithm DETECT_TERMINATION is used in a parallel primary computation on P PEs to detect its termination. */
Begin

PE i, $0 \leq i < P$, executes the following steps:

1. Initialization:
   $idle := 0$; $free := 0$; $inactive := 0$; $\forall$ children j of i, $child\_inactive[j] := 0$; $num\_unack\_msgs := 0$; $terminated = 0$.
   /* Here $idle = 1$ when PE i is idle, else it is 0 when i is busy; $free = 1$ when i is free, else it is 0 when i is loaded; $inactive = 1$ when i is inactive, else it is 0 when i is active; $child\_inactive[j] =$ (number of STOP messages) - (number of RESUME messages) received from child j of i; num_unack_msgs stores the number of primary messages sent out by i that are yet to be acknowledged; and $terminated = 1$ when i has detected termination, else it is 0. The terms parent and child are used with reference to the termination tree. */

Repeat

2. **On** (PE i becoming idle) $idle := 1$.
3. **If** ($idle = 1$) **and** ($num\_unack\_msgs = 0$) **then** $free := 1$.
4. **On** (receiving a STOP from PE j) $child\_inactive[j] := child\_inactive[j] + 1$.
5. **If** ($child\_inactive[j] = 1 \; \forall$ children j of i) **and** ($free = 1$) **and** ($inactive = 0$) **then begin**
   If ($i = root$) **then begin**
   Send TERMINATION to all child PEs;
   $terminated := 1$;
   Endif
   Else Send STOP to parent PE;
   $inactive := 1$;
   Endif
6. **On** (issuing a primary message $M_{i,j}$) $num\_unack\_msgs := num\_unack\_msgs + 1$.
   /* The notation $M_{i,j}$ implies that the sender of the message is PE i and the recipient is PE j. */
7. **On** (receiving a primary message $M_{j,i}$) **begin**
   $idle := 0$;
   $free := 0$;
   If ($inactive = 1$) **then begin**
   Send RESUME$_{j,i}$ to parent PE;
   $inactive := 0$;
   Endif
   Else Send ACKNOWLEDGE$_{j,i}$ to PE j;
   Endon
8. **On** (receiving RESUME$_{j,k}$ from PE l) **begin**
   $child\_inactive[l] := child\_inactive[l] - 1$;
   If ($inactive = 1$) **then begin**
   Send RESUME$_{j,k}$ to parent PE;
   $inactive := 0$;
   EndIf
   Else Send ACKNOWLEDGE$_{j,k}$ to the child PE on the path to PE k;
   Endon
9. **On** (receiving ACKNOWLEDGE$_{j,k}$)
   If ($i = j$) **then** $num\_unack\_msgs := num\_unack\_msgs - 1$;
   Else if ($i = k$) **then** Send ACKNOWLEDGE$_{j,k}$ to PE j;
   Else Send ACKNOWLEDGE$_{j,k}$ to the child PE on the path to PE k.
10. **On** (receiving TERMINATION) **begin**
    Send TERMINATION to all child PEs;
    $terminated := 1$;
    Endon

Until ($terminated = 1$)

End /* Algorithm DETECT_TERMINATION */

Figure 1: Algorithm for termination detection

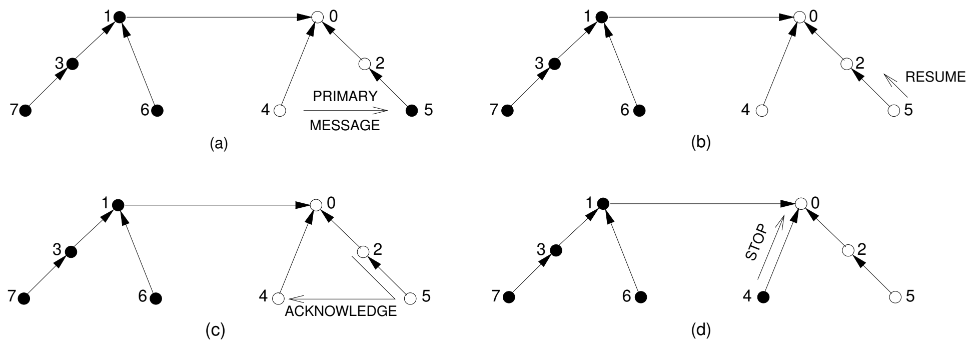


Figure 2: Illustration of the termination detection procedure on an arbitrary architecture. (a) PE 4 sends a primary message to PE 5. (b) PE 5 resumes and sends a RESUME message up the termination tree to PE 2. (c) PE 2 sends an ACKNOWLEDGE message to PE 4 via PE 5. (d) PE 4 relinquishes ownership of the previously sent primary message, and reports a STOP (assuming it is free). White circles denote active PEs and black circles denote inactive PEs.

2.3 Algorithm Illustration

In Fig. 2, we show a spanning tree mapped onto some target architecture to illustrate the above termination detection procedure. Black circles represent inactive PEs that have received STOP messages from all their child PEs and have also reported a STOP message to their corresponding parent PEs (after reporting the last RESUME, if at all they reported one); white circles represent active PEs that have not yet reported a STOP to their parent PEs (after reporting the last RESUME, if at all they reported one). Thus in Fig. 2(a), all PEs except PEs 0, 2, and 4 have reported a STOP message to their corresponding parent PEs. At this time, PE 4 sends a primary message to neighboring PE 5. Since PE 4 owns this message until it receives a corresponding ACKNOWLEDGE message, it is not allowed to report a STOP even if it becomes idle in the meantime. Note that if PE 4 is allowed to report a STOP at this point, and if PEs 0 and 2 also stop before the primary message is received by PE 5, then an incorrect termination will be signaled by PE 0—the primary-computation load corresponding to the primary message from PE 4 to PE 5 has not been processed at this point. Therefore we require all primary messages to be acknowledged before a PE can report a STOP. When the message sent by PE 4 is received by PE 5, it resumes and sends a RESUME message up the termination tree to nullify a previously transmitted STOP message along this path (see Fig. 2(b)). When the RESUME message is received by PE 2, it no longer needs to be transmitted any further up the tree, since there is no prior STOP message to be neutralized. So an ACKNOWLEDGE message is transmitted from PE 2 to PE 4 via PE 5, to acknowledge the primary message received by PE 5 (see Fig. 2(c)). On receiving the ACKNOWLEDGE message, PE 4 relinquishes ownership of the primary message previously sent to PE 5 and can report STOP if it is idle, and does not own any other primary messages (see Fig. 2(d)).

3 Proof of Correctness

3.1 Preliminaries

Before proving the correctness of algorithm DETECT_TERMINATION, we need to define a few terms. All algorithm statements cited hereafter refer to the statements in DETECT_TERMINATION. Let t_0 denote the time at which the initialization of termination-detection variables (statement 1) is completed; the primary computation begins after t_0. At any time $t \geq t_0$, a PE is either active or inactive. By Def. 3, the root PE is active until it signals termination.

Definition 4 The *state* of the termination tree at any time $t \geq t_0$ is defined by the set $\mathcal{A}$ of active PEs at that time, where t_0 is the time the primary computation begins.

The state changes either during a *STOP* or a *RESUME event*; these terms, and the related concepts of *STOP* and *RESUME paths* are defined as follows.

Definition 5 A *STOP event* at an active non-root PE u_0 is said to *start* when it becomes inactive and reports a STOP to its parent, or in the case of the root PE, when it reports TERMINATION. The *STOP path* corresponding to u_0 for this event is the maximal ordered set of PEs $(u_0, u_1, \ldots, u_i)$ such that each PE in the set is the parent of the preceding PE, and at the beginning of the STOP event, u_j is free and waiting for at most one STOP, for $0 \leq j \leq i$. The STOP event is said to be *complete* when the STOP from u_0 either 1) reaches a PE on the STOP path that is not free or is waiting for at least one other STOP (if, in the meantime, it received a RESUME from a child not on the STOP path), or 2) reaches the parent of u_i—if u_i is the root PE, then the STOP event is complete when the root PE signals TERMINATION.

Definition 6 A *RESUME event* at an inactive PE v_0 is said to *start* when it receives a primary message, becomes active, and reports a RESUME to its parent. The *RESUME path* corresponding to v_0 for this event is the maximal ordered set of PEs $(v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_j)$ such that each PE in the set is the parent of the preceding PE, and at the beginning of the RESUME event, PEs v_k, $0 \leq k \leq i$, are inactive, and PEs v_l, $i+1 \leq l \leq j$, are active (v_j is an active PE that does not have an active parent). The RESUME event is said to be *complete* when the RESUME from v_0 reaches a PE on the RESUME path that is active at the time it receives the RESUME.

Definition 7 A *state event* is either a STOP or a RESUME event and has the same start and completion times as the STOP or RESUME event. The start and completion times of a state event a are denoted by $t_s(a)$ and $t_c(a)$, respectively.

We denote the time at which the mth state event is complete after t_0 by t_m. It is necessary in the correctness proofs to formalize the notion of a set of simultaneous state events, which we do next.

Definition 8 Two state events a and b are said to be *simultaneous* if $t_c(b) > t_s(a)$ and $t_c(a) > t_s(b)$.

Figure 3(a) shows three STOP events a, b, and c and a RESUME event d along with their temporal relationships. In this example, all pairs of events are simultaneous except pairs (a, c) and (b, c), since STOP event c can take place only when both pending STOPs corresponding to events a and b have been received.

Definition 9 Given a set of events $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$, an *overlap graph* of $\mathcal{E}$ is one in which the vertices are the events in $\mathcal{E}$ and the edges are those connecting simultaneous events.

Figure 3(b) depicts the overlap graph for the events in Fig. 3(a).

Definition 10 A set of events is said to be *simultaneous* if its overlap graph is connected.

Thus from Fig. 3(b), the set of events in the above example are simultaneous.

Next, we prove that DETECT_TERMINATION does not incorrectly signal termination.

3.2 Proof of No Incorrect Termination Detection

First, in Lemma 1, we show how STOP and RESUME events affect the state of the termination tree when message communication in links is in FIFO order. Since STOP and RESUME messages from a PE alternate, it suffices in this case to just set to 1 and reset to 0 the elements of *child_inactive* in statements 4 and 8, respectively. Using this lemma, we show next in Theorem 2 that DETECT_TERMINATION does not incorrectly signal termination when FIFO communication takes place. Finally, in Theorem 3 we show that using increment and decrement operations on the elements of *child_inactive* correctly handles the non-FIFO case as well.

3.2.1 FIFO Case

Lemma 1 *Assume that message communication in links is in FIFO order and that the elements of child_inactive in statements 4 and 8 of DETECT_TERMINATION are set to* 1 *and reset to* 0*, respectively. Then the state of the termination tree at the completion of a set of multiple simultaneous state events is the same as when these state events occur one after another in order of their start times, with ties broken arbitrarily.*

Proof: First, consider a single state event corresponding to a single STOP or a single RESUME event. When only a single STOP event occurs, say, at PE i, PEs on the STOP path will become inactive one after the other as the STOP from i goes up the termination tree (statements 4 and 5), since from Def. 5 all PEs on the path are free and waiting for just one STOP (except i which is not waiting for any); if the STOP goes beyond the STOP path, then that implies another STOP event. When only a single RESUME event occurs, say, at PE j, then all PEs on the RESUME path will be active after the event, since the RESUME from j goes up the tree until it reaches an active ancestor of j (statements 7 and 8). Note that in the above discussion, we have assumed FIFO communication so that the STOP along the STOP path and the RESUME along the RESUME path reach PEs after any previously sent RESUME and STOP along that path, thus making PEs on the path inactive and active, respectively. We use this assumption similarly in the rest of the proof.

Now consider the effect of a set of simultaneous state events on the state of the termination tree. We prove the lemma by showing that the effect of simultaneous STOP and RESUME events is unique regardless of the rate at which STOPs and RESUMEs corresponding to the events traverse the various STOP and RESUME paths (as long as they are in FIFO order). Therefore a given ordering of these events, for instance, by start times, in which event b follows event a will simply correspond to the scenario in which the STOP or RESUME associated with b waits at its original PE till that associated with a has traversed as far as it can along its path. Since STOP and RESUME events affect PEs on only their paths, events with nonintersecting paths will not interact

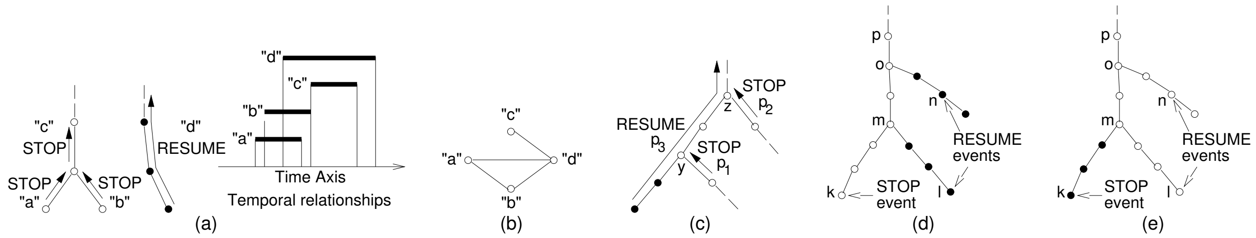


Figure 3: Illustration of concepts in the proof of Lemma 1: (a) Set of three STOP events a, b, and c and a RESUME event d, and their temporal relationships. (b) Overlap graph for the set of events in [a]. (c) The impossibility of two STOP paths p_1 and p_2 intersecting a RESUME path p_3. (d) Illustration of the effect of a STOP and multiple RESUME events, all simultaneous, and all with intersecting paths on a part of the termination tree before the events start. (e) The same scenario as in [d] but after the events complete. White circles denote active PEs and black ones inactive PEs.

with each other, and hence after these events, PEs on STOP paths will be inactive, and those on RESUME paths active. No two STOP paths can intersect, since, otherwise, it would mean that the first PE u where they intersect has two pending STOPs, which is not possible since u lies on a STOP path. RESUME paths that intersect only other RESUME paths will have all PEs on them active after the corresponding RESUME events, since a RESUME goes up along each path until it reaches an active PE.

Next, we consider intersecting STOP and RESUME paths. From the definition of a RESUME path in Def. 6, if two RESUME paths first intersect at a PE x, then all ancestor PEs of x that are on the first path will also be on the second. Therefore, if there is a set of RESUME paths in which every path intersects at least one other path, then there will be some PE common to all paths, so that all paths will intersect each other. Consider now two STOP paths p_1 and p_2 intersecting a RESUME path p_3 at PEs y and its ancestor z, respectively, as shown in Fig. 3(c). Then, from the definition of a STOP path in Def. 5, both y and z are active and waiting for just one STOP, and from the definition of a RESUME path, all PEs on the path from y to z are active. This, however, is contradictory since it implies z is waiting for two STOPs, i.e., it has two active children, one along p_2 and one along the RESUME path from y. Therefore, no two STOP paths can intersect the same RESUME path.

Thus, the only remaining possibility to consider is that of a STOP path intersecting one or more RESUME paths. Figure 3(d) depicts such a case in which a STOP event occurs at PE k with STOP path $(k, \ldots, m, \ldots, p, \ldots)$, and RESUME events occur at PEs l and n with RESUME paths $(l, \ldots, m, \ldots, o, p, \ldots)$ and $(n, \ldots, o, \ldots, p, \ldots)$, respectively. The STOP from PE k and the RESUMEs from the PEs where RESUME events occur (PEs l and n in the above example) will go up the tree making PEs inactive and active on their way, respectively, as in the single state-event case, except for the following difference. The STOP event will complete when the STOP from PE k: 1) has traversed the complete STOP path and hence made all PEs on the path inactive, or 2) when it is blocked by a PE on the STOP path that also lies on some RESUME path and that received at least one RESUME from the RESUME paths that it lies on before it received the STOP. The blocking occurs because the PE in question has STOPs pending along the paths from which it received RESUMEs (statements 8 and 5).

The RESUMEs from the PEs where RESUME events occurred will go up the tree making PEs active on their way until they reach a PE that is already active. Note that from the definitions of STOP and RESUME paths in Defs. 5 and 6 it follows that if a STOP path first intersects a RESUME path at some PE x, then PE x and all its ancestors that lie on the STOP path will also lie on the RESUME path—this is true even when the STOP path is a subset of a RESUME path. It also follows from these definitions that a RESUME path cannot be a subset of the STOP path. Therefore, if the STOP from PE k made some PEs lying on any RESUME path inactive, then such PEs will later receive a RESUME from one of the RESUME paths they lie on and will become active. Therefore, after all STOP and RESUME events complete, regardless of the rate at which the STOP and RESUMEs travel, PEs on all RESUME paths will be active, and PEs on the subpath of the STOP path that is disjoint from all RESUME paths will be inactive. This is illustrated in Fig. 3(e). Since in all cases we obtain a unique effect, simultaneous STOP and

RESUME events can be ordered as mentioned in the lemma without changing the final state of the termination tree. □

Theorem 2 *Assuming message communication in links is in FIFO order and that the elements of child_inactive in statements 4 and 8 of DETECT_TERMINATION are set to* 1 *and reset to* 0*, respectively, DETECT_TERMINATION does not signal termination when there is primary computation to be performed.*

Proof: First, we show that between state events, the set $\mathcal{A}$ of active PEs induces a tree, the *active tree* $\mathcal{T}$, rooted at the root PE in the termination tree, and that at all times, the root PE is active if there are any other active PEs. Recall from Lemma 1 that simultaneous STOP and RESUME state events can be ordered one after another by their start times, (with ties broken arbitrarily) without altering the final state of the termination tree. Therefore, we assume in the rest of the proof that simultaneous state events are ordered in this manner, so that at any time only a single STOP or RESUME state event takes place. We first establish the hypothesis below concerning the active tree using induction on the number of state events.

Hypothesis: Between any two state events, the set $\mathcal{A}$ of active PEs induces a tree, the *active tree* $\mathcal{T}$, rooted at the root PE in the termination tree. Moreover, at any time, if $\mathcal{A} \neq \emptyset$, then $root \in \mathcal{A}$.

Induction Basis: At t_0, the base case when the termination tree has not undergone any state changes, the hypothesis is vacuously true as all PEs are active and the active tree is the termination tree.

Induction Step: We assume that the hypothesis holds at time t_m (just after the mth state event completes), and consider its validity at time t_{m+1} (just after the $(m+1)$th state event completes)—since there are no state changes after the mth and before the $(m+1)$th state events, the hypothesis remains valid during that interval; we will show that the second part of the hypothesis concerning the root PE also holds during the $(m+1)$th state event. The $(m+1)$th state event will be either a STOP or a RESUME event. From the proof of Lemma 1, we know that in a STOP event, active PEs on the STOP path become inactive one by one from the first to the last PE on the path, and that in a RESUME event, inactive PEs on the RESUME path become active from the first to the last inactive PE on the path.

Suppose the $(m+1)$th state event is a STOP event. There are two facts to consider. First, the set of active PEs $\mathcal{A}_m$ at t_m induces an active tree $\mathcal{T}_m$ rooted at the root PE. Second, from the definition of a STOP path in Def. 5, the only active child of a PE on the path is also on this path (otherwise, the PE will have two pending STOPs), except for the first PE which has no active children, and hence is a leaf of $\mathcal{T}_m$. These two facts combined imply that when PEs on the STOP path become inactive at t_{m+1} and are deleted from $\mathcal{T}_m$, the rest of the active PEs will remain connected to form an active tree $\mathcal{T}_{m+1}$ rooted at the root PE. It also follows that if the root PE is on the STOP path—the only way it can become inactive at t_{m+1}—then its only active child, if any, at t_m is on the STOP path. This active child in turn can have at most one active child which must also lie on the STOP path, and so on, until we reach the first PE on the STOP path which is a leaf of $\mathcal{T}_m$. That is, if the root PE is on the STOP path, then there is a string of active PEs from the root PE to a leaf of $\mathcal{T}_m$, such that all their children not on the path are inactive at t_m. Since there is an active tree at t_m, this implies that $\mathcal{T}_m$ is the STOP path. Hence, if the root PE becomes inactive at t_{m+1}, all other active PEs will become inactive before it. Thus the hypothesis holds for a STOP event.

Next, consider a RESUME event. From the definition of a RESUME path in Def. 6, the parent of the last inactive PE on the RESUME path is a leaf of $\mathcal{T}_m$. Therefore, when inactive PEs on the RESUME path become active at t_{m+1}, $\mathcal{T}_m$ will simply have a path of active PEs attached to a leaf PE, so that $\mathcal{T}_{m+1}$ will also be an active tree rooted at the root PE. Since no PE becomes inactive, the root PE will be active throughout the RESUME event. Hence the hypothesis holds for a RESUME event too. Figures 4(a) and (b) depict how the active tree in an arbitrary termination tree at t_m changes to that at t_{m+1} due to simultaneous STOP and RESUME events, the effect of

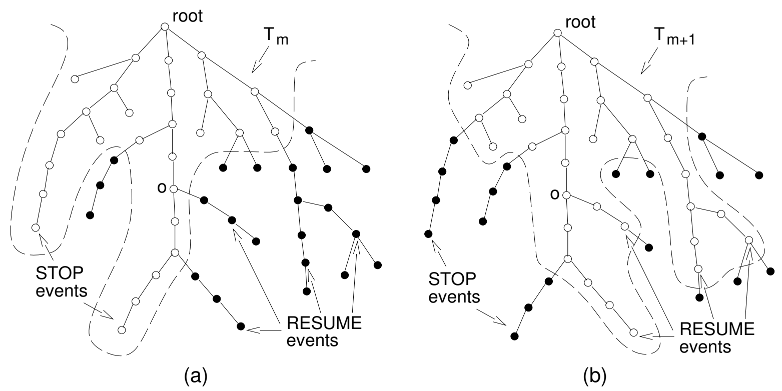


Figure 4: (a) State of the termination tree before the $(m+1)$th state event starts showing the active tree $\mathcal{T}_m$ at that time, and (b) State of the termination tree after the $(m+1)$th state event completes showing the new active tree $\mathcal{T}_{m+1}$. White circles denote active PEs and black ones inactive PEs. The dashed curve demarcates the active tree.

events being obtained by ordering them in an arbitrary manner.

Note that all primary-computation load is either with active PEs, or is in primary messages which in turn imply at least one active PE (because of pending acknowledgements). This means that if there is primary computation to be performed, then $\mathcal{A} \neq \emptyset$. Moreover, since from the above hypothesis, $root \in \mathcal{A}$ if $\mathcal{A} \neq \emptyset$, and since the root PE will not signal termination when it is active, DETECT_TERMINATION will not signal termination when there is primary computation to be performed, thus proving Theorem 2. □

3.2.2 Non-FIFO Case

Theorem 3 *DETECT_TERMINATION does not signal termination when there is primary computation to be performed, irrespective of whether message communication in links is in FIFO order or not.*

Proof: STOP and RESUME messages from any PE j must alternate and the first message among these two types must be a STOP message. Therefore at any time, $0 \leq$ (number of STOPs sent out by j) − (number of RESUMEs sent out by j) ≤ 1. Also, note that *child_inactive*$[j]$ in parent i of j records the number of STOPs minus the number of RESUMEs that i has received from j at any time. Thus, *child_inactive*$[j] = 1$ for child j of PE i implies that (1) if all STOP and RESUME messages sent by PE j have been received by PE i, then the last message sent by PE j was a STOP message, or (2) else there is at least one RESUME message sent by PE j that has not yet been received by PE i. In the first case, *child_inactive*$[j]$ correctly reflects the state of PE j, i.e., PE j has no computations to perform, and hence PE i can report a STOP to its parent once all its children have reported a STOP (i.e., *child_inactive*$[k] = 1$, $\forall$ children k of i) and it has become free. This will result in correct termination detection. In the second case also, PE i can report a STOP to its parent once the above two conditions are satisfied. However, this will not result in signaling of termination since the sender PEs corresponding to the pending RESUME messages (and hence pending acknowledgements) will not have reported a STOP. When these RESUME messages are later received, they will be passed up the termination tree to nullify previously sent STOP messages.

Hence we see that by having increment-decrement operations on the *child_inactive* array we are able to achieve the same effect as when message communication is FIFO, albeit with some intermediate spurious changes which are corrected after some delay. Note that we also showed that during these spurious changes (that do not occur when message communication is FIFO) termination is not signaled. Thus the theorem follows from Theorem 2. □

3.3 Proof of Finite Detection Delay

Next we show that our algorithm has a finite detection delay; we will analyze its best- and worst-case detection delays in Sec. 4.

Theorem 4 *DETECT_TERMINATION signals termination a finite time after the primary computation is complete.*

Proof: After the primary computation is complete, say, at time t_a, and after all pending acknowledgements been received, say, at time t'_a, there are no more primary messages sent or received, hence inactive PEs can no longer become active. Therefore the active tree $\mathcal{T}_a$ at time t'_a can either contract towards the root PE or remain unchanged. An active PE that completes its computation at time t_a will have $free = 1$ once it has received any pending acknowledgements (statement 3), which it will in finite time since ACKNOWLEDGE messages follow a definite finite route. The leaves of the active tree are PEs with all inactive children. These leaf PEs will therefore report a STOP after they have become free (statement 5). Next, when the new set of leaf PEs in the active tree receive these STOPs, they will report a STOP to their parents and become inactive, so that the active tree contracts again towards the root PE. Since the termination tree has a finite-length root-to-leaf path (or finite height), the root PE will eventually become inactive a finite time after t_a and signal termination. □

Theorem 5 *DETECT_TERMINATION correctly detects termination of any parallel primary computation, irrespective of whether messages in links are communicated in FIFO order or not.*

Proof: From Theorems 2 and 3, DETECT_TERMINATION will not signal termination incorrectly if the primary computation is not yet completed. The rest of the proof follows from Theorem 4. □

4 Performance Analysis

Here we analyze the performance of our termination detection algorithm and suggest modifications that can improve its average-case performance.

4.1 Detection Delay

Before we analyze the detection delay, we need to define a few terms in relation to a rooted tree. A vertex is said to be at *level* i if it is at a distance of i from the root [7]. Thus the root is at level 0. The *height* of a vertex is defined recursively in terms of those of its children as follows. The height of leaf vertices is 0. The height of an internal vertex is one more than the maximum height of its children. Note that when all leaf PEs, which are height at 0, are deleted from the tree, the new set of leaf PEs are those originally at height 1. Similarly, when these leaf PEs are deleted from the tree, the new set of leaf PEs are those originally at height 2, and so on. The *height* of the tree is the maximum height among all vertices, which is equal to the height of the root or the level of the leaf vertex at the maximum level. The *ith-height degree* of the tree is the maximum number of children at height i that any vertex has.

Next, we establish the best- and worst-case detection delays of our algorithm. We will see that in the worst-case of detection delay, the active tree is the complete termination tree at termination (and after all pending acknowledgements have been received), and that STOP messages are passed up the tree in the same manner as in an *all-to-one accumulation* operation at the root PE. Thus each PE receives STOP messages from all its child PEs and forwards a single STOP to its parent PE. For the worst-case detection delay complexity, we derive a general expression, and also a simplified one in which we assume that the total STOP-message communication time after the primary computation terminates dominates the total STOP-message processing time. This is a realistic assumption, since processing of a STOP message involves an increment operation (statements 4) which can be completed in a single instruction cycle on most systems, compared to the several cycles it may take to communicate a STOP message to a parent PE (see also footnote 6).

Theorem 6 *DETECT_TERMINATION has a best-case detection delay of* $\Theta(1)$ *and, a finite optimal worst-case detection delay on the same order as an optimal one-to-all broadcast (or all-to-one accumulation) on the target topology of* $\Theta(D+\sum_{i=0}^{H-1} s_i)$*, where* D *is the diameter of the target architecture,* $H = \Theta(D)$ *the height of the termination tree, and* s_i *the* ith*-height degree of the termination tree. If the total STOP-message transmission time dominates the total STOP-message processing time in the interval between primary computation termination and its subsequent detection, then the worst-case detection delay is* $\Theta(D)$*.*

Proof: From the proof for Theorem 4, after the primary computation is complete at time t_a, and after all pending acknowledgements have been received at time t'_a, the active tree $\mathcal{T}_a$ at time t'_a can only contract towards the root PE. In the best case, the root PE will be the only active PE at time t_a and will obviously have no pending acknowledgements. Therefore, it will signal termination in $\Theta(1)$ time as soon as it becomes idle. In the worst case, all PEs will be active at time t'_a and thereafter, the active tree $\mathcal{T}_a$ will contract towards the root PE. Let t''_a denote the time after t'_a by which $\mathcal{T}_a$ contracts to an empty set, i.e., t''_a is the time termination is detected. We will show that, in the worst case, RESUME and ACKNOWLEDGE messages are passed up and down the termination tree between t_a and t'_a as messages in an all-to-one accumulation operation at the root PE and in an one-to-all broadcast operation from the root PE, respectively. We will also show that the active-tree contraction process between t'_a and t''_a is similar to a all-to-one accumulation operation at the root PE, where the messages involved are STOP messages. Thus, $t'_a - t_a = \Theta(2 \cdot (t''_a - t'_a))$ in the worst case, and $T_d = t''_a - t_a = \Theta(t''_a - t'_a)$.

First, consider active-tree contraction between t'_a and t''_a. In the worst case, the active tree at t'_a will be equal to the termination tree. Therefore at t'_a, the leaf PEs of the active tree, which are at height 0, will report a STOP and become inactive. Next, when the new set of leaf PEs in the active tree, which are height at 1, receive these STOPs at time $t_b = t'_a + \tau_c$, they will report a STOP to their parents and become inactive at time $t'_b = t_b + s_0\tau_m = t'_a + \tau_c + s_0\tau_m$, so that the active tree contracts by one level towards the root PE; here s_0 is the 0th-height degree of the termination tree and is equal to the maximum number of STOPs received (and processed) by any PE at time t_b, and τ_c is the time to transmit a STOP message over a single communication link and τ_m the time to process a single STOP message. Similarly, at time $t_c = t'_b + \tau_c = t'_a + 2\tau_c + s_0\tau_m$ when the new leaves of the active tree at height 2 receive these STOPs, they will report a STOP to their parents and become inactive at time $t'_c = t_c + s_1\tau_m = t'_a + 2\tau_c + (s_0 + s_1)\tau_m$, so that again the active tree contracts by one level towards the root PE; here s_1 is the 1st-height degree of the termination tree and is equal to the maximum number of STOPs received (and processed) by any PE at time t_c. Thus, STOP messages are passed up in the termination tree during active-tree contraction in the same manner as messages in an all-to-one accumulation operation at the root PE. Since there are H stages of contraction, active-tree contraction time $t''_a - t'_a = H\tau_c + (\sum_{i=0}^{H-1} s_i)\tau_m = \Theta(D + \sum_{i=0}^{H-1} s_i)$

(since $H \leq D$ and τ_c and τ_m are constants of the target system), where s_i is the ith-height degree of the termination tree or the maximum number of STOPs received (and processed) by any PE at the ith stage of active-tree contraction.

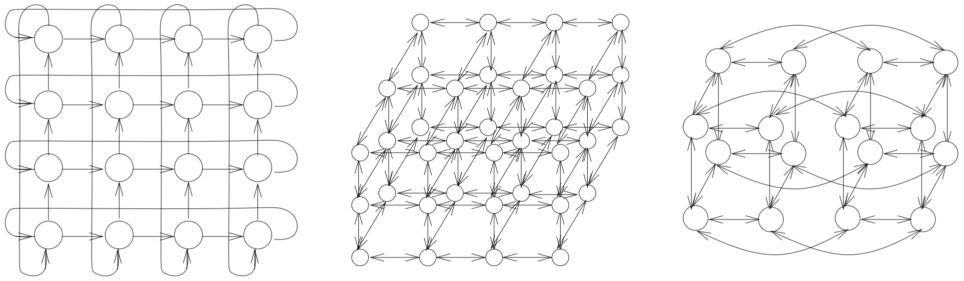
Next, we consider the time $t'_a - t_a$ in the worst case required for guaranteeing receipt of all acknowledgements after termination. Since STOP and RESUME messages from a PE must alternate, at any time, at most one RESUME message will need to be sent from a PE to its parent, so that different RESUME messages will not contend for the same communication link. Moreover, since a RESUME and the corresponding ACKNOWLEDGE message travel in opposite directions along the same path (statements 8 and 9), there also cannot be contention for communication links between different ACKNOWLEDGE messages. Clearly, in the worst case, all PEs except the root and leaf PEs will be inactive just before t_a, and the leaf PEs will all send a RESUME up the tree (after receiving some primary message), and just after that finish their computations and become idle at t_a. These RESUME messages will reach PEs at height 1 in the termination tree, which in turn will report RESUMEs. Thus RESUME message passing in the worst case is similar to a all-to-one accumulation operation at the root PE. After these RESUME messages reach the root PE, ACKNOWLEDGE messages will come down the termination tree in the opposite direction from the root PE to leaf PEs. Thus in the worst-case, ACKNOWLEDGE message passing resembles a one-to-all broadcast from the root PE, which takes the same time as an all-to-one accumulation operation. Therefore the time $t'_a - t_a$ required to guarantee receipt of all ACKNOWLEDGE messages is $O(t''_a - t'_a)$. Hence the worst-case detection delay is $\Theta(t''_a - t'_a) = \Theta(D + \sum_{i=0}^{H-1} s_i)$.

Note that in the worst-case, for any algorithm, information about the states of all PEs will need to be collected after time t_a for termination detection. This happens when all PEs simultaneously complete their computation at time t_a. To detect termination in this case, we essentially need a all-to-one accumulation operation at the root PE. An optimal all-to-one accumulation operation at the root PE can be performed by reversing the communication paths in an optimal one-to-all broadcast from the root PE [19]. Therefore the worst-case detection delay for any termination detection algorithm is at least the time for an optimal broadcast. Recall from Sec. 2 that the termination tree is chosen so as to optimize a one-to-all broadcast from the root PE. Since in the worst case DETECT_TERMINATION performs three broadcast- or accumulation-type operations after termination and before detection, its worst-case detection delay is optimal for any topology and is on the same order as an optimal one-to-all broadcast on that topology.

Normally, $\tau_c \gg \tau_m$, so that the communication delay, given by the $H\tau_c$ term, will dominate the time $(\sum_{i=0}^{H-1} s_i)\tau_m$ to process STOP messages, in which case the worst-case detection delay is $\Theta(H) = \Theta(D)$. $\Box$

An interesting offshoot of the above analysis is that it has led to a general expression for one-to-all broadcast in terms of the diameter and height degrees of the target topology, which seems to be a new result.

In the above theorem, we determined detection delay assuming message passing takes place only between neighbors. When message passing takes place between arbitrary PEs, the ACKNOWLEDGE messages from different active ancestor PEs of previously inactive recipient PEs are transmitted to possibly non-neighboring sender PEs. Hence, there can be contention between different ACKNOWLEDGE messages that are sent after time t_a, and thus they may take more time than $\Theta(H) = \Theta(D)$ time to reach the sender PEs, after which the active tree starts contracting monotonically as before. However, in most practical cases, this contention will be negligible, firstly, because the number of ACKNOWLEDGE messages after time t_a will likely be small, and, secondly, because each ACKNOWLEDGE message is a very short constant-length message. Therefore, even when message passing takes place between arbitrary PEs, the worst-case detection delay will most likely be $\Theta(D)$.



(a) (b) (c)

Figure 5: Examples of k-ary n-cube tori: (a) 2-D torus, (b) 3-D mesh, and (c) hypercube.

4.1.1 Detection Delay for k-ary n-cubes

Here we simplify the worst-case detection delay expression for DETECT_TERMINATION for an interesting and very general class of architectures called k-ary n-cube tori that includes rings ($n = 1$), 2-D tori ($n = 2$), 3-D tori ($n = 3$) and hypercubes ($k = 2$) [6] (see Fig. 5), by determining the time for an optimal broadcast. Here, n is referred to as the dimension and k the radix of the architecture. Every processor has an n-digit radix-k label $a_{n-1}a_{n-2}\ldots a_i \ldots a_1a_0$, and has neighbors $a_{n-1}a_{n-2}\ldots((a_i+1) \bmod k)\ldots a_1a_0$ and $a_{n-1}a_{n-2}\ldots((a_i-1) \bmod k)\ldots a_1a_0$ along each dimension i. A k-ary n-cube tori consists of k k-ary $(n-1)$-cube tori with corresponding PEs in each k-ary $(n-1)$-cube connected in a ring. The diameter of a k-ary n-cube is $nk/2$. Another class of architectures called k-ary n-cube meshes is also very general, and differ from the k-ary n-cube tori class only in that they do not have end-around connections in any dimension; the linear array ($n = 1$), 2-D mesh ($n = 2$), and 3-D mesh are its special cases (see Fig. 5).

Corollary 1 *DETECT_TERMINATION has an optimal worst-case detection delay on k-ary n-cube tori and meshes of $\Theta(D)$, where D is the diameter of the architecture.*

Proof: An optimal broadcast in a k-ary n-cube tori can be performed, based upon its recursive construction from a single PE, as follows. Let the root PE, which has the data to be broadcast, have label 0^n.[3] There are in all n stages. In each stage i, for $1 \leq i \leq n$, all PEs with labels of

[3] Superscripts in labels denote concatenation.

the form $0^{n-i+1}x$ that already have the data, broadcast it on a ring of k PEs including PEs with labels of the form $0^{n-i}bx$; here x is any $(i-1)$-digit radix-k label and $1 \leq b < k$. Since each such stage takes $\Theta(k/2)$ time, and since there are n stages in all, the time for optimal broadcast on a k-ary n-cube tori is $\Theta(nk/2) = \Theta(D)$; essentially, in the $(\sum_{i=1}^{d-1} s_i)\tau_m$ message-processing time for optimal broadcast, the s_i's are all unity. Similarly, one can establish the same result for k-ary n-cube meshes. □

4.2 Message Complexity

We now consider the message complexity of DETECT_TERMINATION. Since each primary message can potentially cause $\Theta(D)$ RESUME messages before an acknowledgement for it is issued, the message complexity can be as much as $\Theta(\max(MD, P))$. However, on the average the message complexity will likely be $\Theta(\max(M, P))$ because a single primary message is likely to cause multiple RESUMEs only towards the end of the primary computation. At all other times, when enough computation load is available with PEs, only a single RESUME is likely to be caused by a primary message. We establish the above claim regarding average message complexity under realistic assumptions in the theorem below.

4.2.1 Average Message Complexity

Theorem 7 *Assuming that all PEs have the same free probability $q(t)$ at any time t in the interval (t_0, t_a), and that $\frac{1}{1-q(t)}$ (i.e., the reciprocal loaded probability) averaged over (t_0, t_a) is bounded above by a constant, the average message complexity $M_{s,avg}$ of DETECT_TERMINATION is $\Theta(\max(M, P))$ on all architectures. Here t_0 is the time when the primary computation begins and t_a the time it terminates, M is the number of primary messages, and P the number of PEs.*

Proof: Secondary messages used by DETECT_TERMINATION are STOP, RESUME, ACKNOWLEDGE, and TERMINATION. The number of STOPs used is at least $\Theta(P)$ since every PE must report a STOP for termination detection. The number of STOPs required beyond this is the same as the number of RESUMEs. The number of TERMINATION messages is $\Theta(P)$. Finally, since the number of links an ACKNOWLEDGE message traverses is just one more than the number of RESUME messages triggered by an inactive recipient PE, the ACKNOWLEDGE-message complexity is the same as the RESUME-message complexity plus $\Theta(M)$ (for M more ACKNOWLEDGEs than RESUMEs). Therefore the average (secondary) message complexity of DETECT_TERMINATION is the maximum of $\Theta(P)$, $\Theta(M)$, and the RESUME-message complexity. We now determine an upper bound on the average number of RESUME messages caused by a single primary message. Consider a PE u_0 in the termination tree of an arbitrary architecture with sequence of ancestors $(u_1, u_2, \ldots, u_{m-1})$, where u_1 is the parent of u_0 and u_{m-1} is the root PE. Let d_i, for $1 \leq i \leq m-1$, denote the number of descendents of u_i in the termination tree. If a primary message is received by PE u_0, it will not cause any RESUME messages if u_0 or any of its descendents is loaded, since in that case u_0 will be active (statement 7). A single RESUME message will result if u_1 is loaded and all its descendents (including u_0) are free. Similarly, i RESUMEs will result, for $1 \leq i \leq m-1$, if u_i is loaded and all its descendents are free. Therefore the average number of RESUMEs resulting from the receipt of a single primary message by PE u_0 is

$$\mathcal{M}_{avg}(u_0) = \sum_{i=1}^{m-1} \left[(1-q) \cdot q^{d_i} \cdot i\right]$$

In the above equation, we have used q instead of $q(t)$ for simplicity. Note that $0 < d_1 < d_2 < \ldots < d_{m-1}$ and $m \leq P$ for any PE on any architecture, and that $0 \leq q(t) < 1$ for $t \in (t_0, t_a)$—$q < 1$ because before t_a the primary computation is not terminated. Therefore, we can replace m by P and q^{d_i} by q^i in the above expression to upper bound the average number of RESUMEs caused by a primary message as:

$$\begin{aligned}
\mathcal{M}_{avg} &\leq \sum_{i=1}^{P-1} \left[(1-q) \cdot q^i \cdot i\right] = (1-q) \cdot q \cdot \sum_{i=0}^{P-1} \left[i \cdot q^{(i-1)}\right] \\
&= (1-q) \cdot q \cdot \frac{d}{dq} \left[\sum_{i=0}^{P-1} q^i\right] = (1-q) \cdot q \cdot \frac{d}{dq} \left[\frac{1-q^P}{1-q}\right] \\
&= q \cdot \frac{1 - q^{(P-1)}\left[P - (P-1)q\right]}{1-q}
\end{aligned}$$

Since both q and $1 - q^{(P-1)}\,[P - (P-1)q]$ are less than one, and since it is assumed that the reciprocal loaded probability $\frac{1}{1-q} = \Theta(1)$ on the average over the time interval of interest, the average number of RESUMEs per primary message is $\mathcal{M}_{avg} = O(\frac{1}{1-q}) = O(1)$, or the total average RESUME-message complexity is $O(M)$. Therefore the average secondary-message complexity of DETECT_TERMINATION under the assumptions of the theorem is $M_{s,avg} = \Theta(\max(M, P))$. □

Note that a loaded PE is always busy except when after sending out some primary message it becomes idle and waits for an acknowledgement. In this case, the PE remains loaded and idle until it receives the pending acknowledgement or it receives a primary message and becomes busy. In practice, the likelihood of a PE being loaded and idle will be very small, so that the loaded probability will essentially be the same as the busy probability. The assumption in the above theorem that the average reciprocal loaded probability (or the average reciprocal busy probability) is bounded above by a constant is a realistic one. This is because, in practice, one desires to maintain a certain minimum efficiency in the utilization of PEs [13], which translates to a minimum acceptable busy probability or a maximum acceptable reciprocal busy probability requirement. Thus the problem size or the amount of primary computation is scaled with the number of PEs used to solve a problem to meet this requirement. Moreover, for most applications the busy probability is likely to be high for most of the primary computation, so that the average reciprocal busy probability and hence the constant associated with the average message complexity of DETECT_TERMINATION is likely to be low. Even in an application with a low busy PE probability of, say, only 20%, the constant associated with the average message complexity will be at most 5. Below we give a method for reducing this constant further.

4.2.2 A Practical Technique for Reducing Message Complexity

The constant associated with the average message complexity of DETECT_TERMINATION can be substantially reduced by using for every PE i a counter $cntr_j$ to store the number of ACKNOWLEDGE messages that need to be sent to each sender PE j. Initially, the counter is initialized to zero. Each time a primary message is received from PE j, $cntr_j$ is incremented. After every t_{int} time interval or just before $cntr_j$ is about to overflow, PE i sends a single RESUME message with the value r_j in $cntr_j$ (if it last reported a STOP and if $cntr_j$ is nonzero) to its parent. Thereafter, it resets $cntr_j$, and increments it on receiving primary messages from PE j as before. The RESUME message from i travels up the termination tree until it reaches an active ancestor PE. The ancestor PE then sends an ACKNOWLEDGE message with the same counter value r_j that the RESUME had down the tree to i. PE i then sends an ACKNOWLEDGE message to PE j along with the value r_j to acknowledge r_j primary messages. Thus the number of RESUMEs and ACKNOWLEDGEs is reduced by a factor of r_j. With the reduction in RESUMEs, the number of STOPs also reduces correspondingly. Since RESUME messages are sent out, if needed, every t_{int} time interval for each sender PE, after the primary computation terminates at time t_a, a RESUME may be delayed by maximum t_{int} time. Therefore the detection delay will increase by an additive factor of t_{int}. We keep $t_{int} = \Theta(D)$, so that the detection delay does not change in order terms. Note that higher the primary-message traffic, more will be the extent of RESUME and ACKNOWLEDGE message combining using counters, and hence greater the reduction in average message complexity.

The above modification will help keep the message complexity of our algorithm low, even at high primary-message traffic, without causing counters to overflow (since counters are flushed before overflow occurs). This is in contrast to overflow problems that may occur at higher traffic in other message-efficient algorithms [2, 14] (see Table 1). However, the use of counters in each PE increases the space complexity; but this is normally not a major concern. Moreover, the increase in space

Algorithms	T_d	T'_d	M_s	$M_{s,avg}$	S_s	C_s	$C_{s,avg}$
Optimal Case	$O(D)$	$O(D)$	$O(\max(M,P))$	$O(\max(M,P))$	$\Theta(P)$	$O(\max(M,P))$	$O(\max(M,P))$
Chandrasekaran and Venkatesan [2]	$O(\max(M,D))$	$O(\max(MD))$	$\Theta(\max(M,P))$	$\Theta(\max(M,P))$	$O(M)$	$\Theta(\max(M,P))$	$\Theta(\max(M,P))$
Lai, Tseng and Dong [14]	$O(H_{max})$	$O(H_{max}D)$	$\Theta(\max(M,P))$	$\Theta(\max(M,P))$	$\Theta(P^2)$	$O(\max(MP,P^2))$	$O(\max(MP,P^2))$
Kumar [11]	$O(P)$	$O(P)$	$O(MP)$	$O(MP)$	$\Theta(P)$	$O(MP)$	$O(MP)$
Our Algorithm	$O(D)$	$O(D)$	$O(\max(MD,P))$	$\Theta(\max(M,P))$ [5]	$\Theta(P)$	$O(\max(MD,P))$	$\Theta(\max(M,P))$ [5]

Table 1: Performance comparison of different termination detection algorithms on an arbitrary architecture in terms of: (1) Detection delay T_d when message passing is between only neighboring PEs,[6] (2) Detection delay T'_d when message passing is between arbitrary PEs,[7] (3) Message complexity M_s, (4) Average message complexity $M_{s,avg}$, (5) Space complexity S_s, (6) Computational complexity C_s, and (7) Average computational complexity $C_{s,avg}$. D denotes the diameter of the target architecture, M the number of primary messages used by the primary computation, P the number of PEs, and H_{max} the height of a maximum-height tree that can be embedded in the target architecture.

complexity per PE is proportional to the number of its senders, and quite frequently the possible set of senders to any PE is restricted to a small set of, usually neighboring, PEs [8, 9, 15]. In cases where the number of senders is large, a dynamic data structure such as a binary search tree ordered by sender PE labels can be used (with some small computational overhead) instead of an array to store the counter values associated with the current set of senders, and thus reduce the memory requirement. Thus we conclude from Theorem 7 and the above discussion that in most practical cases, the average message complexity of DETECT_TERMINATION will be $\Theta(\max(M,P))$ with a very small constant associated with it.

[5] Here we have made the realistic assumption that the reciprocal busy probability averaged over the duration of the primary computation is bounded above by a constant (see Theorem 7).

[6] Here we have made the realistic assumption that the total message-transmission time dominates the total message-processing time. The time τ_m to process a single STOP, RESUME, or ACKNOWLEDGE message is basically that required for an increment or decrement operation (statements 4, 8, or 9, respectively), which on most systems can be completed in a single instruction cycle. The time τ_c to transmit such a message basically consists of a startup time (which includes the time to prepare the message—adding header, trailer, and error correction information—the time to determine a route, and the time to establish an interface between the PE and the router chip) and a transfer time (which depends upon the distance between communicating PEs, the network bandwidth—the contribution of these factors are more in distributed systems—the network load and the message length) [12]. Due to all these factors, τ_c may be several cycles and will easily dominate τ_m on practical systems.

[7] Here we have made the realistic assumption that contention between secondary messages after the primary computation has terminated is negligible.

Algorithms	T_d / T'_d	k-ary n-cube Torus or Mesh	Linear Array or Ring	2-D Mesh or Torus	3-D Mesh or Torus	Hypercube
Optimal Case	T_d	$O(nk)$	$O(P)$	$O(P^{1/2})$	$O(P^{1/3})$	$O(\log P)$
	T'_d	$O(nk)$	$O(P)$	$O(P^{1/2})$	$O(P^{1/3})$	$O(\log P)$
Chandrasekaran	T_d	$O(\max(M, nk))$	$O(\max(M, P))$	$O(\max(M, P^{1/2}))$	$O(\max(M, P^{1/3}))$	$O(\max(M, \log P))$
and Venkatesan [2]	T'_d	$O(\max(Mnk))$	$O(\max(MP))$	$O(\max(MP^{1/2}))$	$O(\max(MP^{1/3}))$	$O(\max(M \log P))$
Lai, Tseng	T_d	$O(P)$	$O(P)$	$O(P)$	$O(P)$	$O(P)$
and Dong [14]	T'_d	$O(Pnk)$	$O(P^2)$	$O(P^{3/2})$	$O(P^{4/3})$	$O(P \log P)$
Kumar [11]	T_d	$O(P)$	$O(P)$	$O(P)$	$O(P)$	$O(P)$
	T'_d	$O(P)$	$O(P)$	$O(P)$	$O(P)$	$O(P)$
Our Algorithm	T_d	$O(nk)$	$O(P)$	$O(P^{1/2})$	$O(P^{1/3})$	$O(\log P)$
	T'_d	$O(nk)$	$O(P)$	$O(P^{1/2})$	$O(P^{1/3})$	$O(\log P)$

Table 2: Performance comparison of different termination detection algorithms on some popular architectures in terms of: (1) Detection delay T_d when message passing is between only neighboring PEs,[6] and (2) Detection delay T'_d when message passing is between arbitrary PEs.[7]

4.3 Space and Computational Complexities

Next, in our algorithm, each PE uses four boolean scalar variables *idle*, $free$, *inactive* and *terminated*, one integer scalar variable num_unack_msgs, and an integer array variable *child_inactive* of size equal to the number of child PEs (see Fig. 1). Thus the space complexity of our algorithm is $\Theta(P)$. Since each message is associated with some (constant) computation, the worst-case and average computational complexities are the same as the worst-case and average message complexities of our algorithm, $\Theta(\max(MD, P))$ and $\Theta(\max(M, P))$, respectively.

4.4 Overall Performance and Comparison with Previous Algorithms

The following theorem summarizes our algorithm's performance:

Theorem 8 *DETECT_TERMINATION has the following complexities for various performance metrics:*
(1) An optimal best-case detection delay of $\Theta(1)$ *and a finite optimal worst-case detection delay equal in order terms to the time of an optimal broadcast in the target architecture—for the class of* k*-ary* n*-cube tori and meshes it is* $\Theta(D)$*;*
(2) Worst- and average-case complexities for both messages and computation of $\Theta(\max(MD, P))$ *and* $\Theta(\max(M, P))$*, respectively; and*
(3) An optimal space complexity of $\Theta(P)$*.*
Here P *is the number of processing elements,* M *is the number of primary messages used by the primary computation, and* D *is the diameter of the target architecture.*

In Table 1, we compare the performance of our algorithm with the optimal case and with three of the most efficient algorithms proposed in the past [2, 11, 14]. Note that the detection delay, which is the most important metric, and space complexity of our algorithm are optimal. The worst-case values of the other two metrics are a of factor of D more than the optimal, which is obviously because of the higher message complexity. However, as proved in Theorem 7 and from the discussion following the theorem, the average message complexity, and hence the average computational complexity, for most applications will be $\Theta(\max(M, P))$ with a small constant factor. Furthermore, the constant factor in this complexity can be greatly reduced further by using counters as discussed in Sec. 4.2. With this modification, our algorithm will have optimal or close to optimal values for all metrics.

Note also that our algorithm has equal or better performance than other algorithms in terms of all four metrics: (1) T_d, T'_d: Our algorithm is better than all other algorithms and has optimal performance. This can also be seen in Table 2 where we have tabulated detection delays for all algorithms on several popular networks. The difference between our algorithm's detection delay and those of others is particularly significant on networks that have a reasonable amount of connectivity. Also, note that the detection delay for other algorithms, except [11], increases greatly when message passing between arbitrary PEs is allowed, while it remains the same for our algorithm (see Sec. 4.1). (2) M_s, C_s: We have shown that the message and computational complexities of our algorithm for most applications is $\Theta(\max(M, P))$ on the average, same in order terms as other message-efficient algorithms [2, 14], but with a very small constant associated with it. The message complexity of [11] (which has the same detection delay as our algorithm on linear arrays and rings, but much more on other networks), however, is higher. (3) Finally, the space complexity of our algorithm is much better than that of [2, 14] and equal to that of [11]. Thus, overall, our algorithm has almost optimal performance that is much better than that of other algorithms.

5 Conclusions

Previous termination detection algorithms while being optimal with respect to one performance metric, are inefficient with regard to other metrics. In particular, all previous methods have a worst-case detection delay, the most important metric, of $\Omega(P)$, where P is the total number of processing elements. This can result in undue idling of computing resources and delay in the utilization of primary computation results; this can be especially acute in a distributed system. In this paper, we presented a novel termination detection algorithm that is optimal or near-optimal with respect to all relevant performance metrics: its detection delay is $O(D)$ (optimal), message and computational complexities $O(\max(MD, P))$ ($\Theta(\max(M, P))$ on the average for most applications—the same as other message-efficient algorithms [2, 14]) and space complexity $\Theta(P)$ (optimal), where M is the total number of primary messages used by the primary computation, and D is the diameter of the target topology. A simple modification using counters greatly reduces the constant factor in the message and computational complexities of our algorithm without causing counter-overflow problems that are present in other algorithms [2, 14]. Furthermore, unlike some previous algorithms [2, 4, 17], our algorithm does not require links to support FIFO message communication for correct operation.

References

[1] G.S. Almasi and A. Gottlieb, "Highly Parallel Computing," *Benjamin/Cummings*, Redwood City, CA, 1994.

[2] S. Chandrasekaran and S. Venkatesan, "A Message-Optimal Algorithm for Distributed Termination Detection," *Jour. of Par. and Distr. Computing*, Vol. 8, pp. 245-252, 1990.

[3] K.M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Comm. ACM*, Vol.24, No.4, pp.198-205, Apr. 1981.

[4] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Comp. Sys.*, Vol.3, No.1, pp.63-75, Feb. 1985.

[5] K.M. Chandy and J. Misra, "How Processes Learn," *Distributed Computing*, Vol. 1, No. 1, pp. 40-52, 1986.

[6] W.J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Trans. on Comp.*, Vol.39, No.6, June 1990.

[7] N. Deo, "Graph theory with applications to Engineering and Computer Science," *Prentice Hall*, N.J., 1989.

[8] S. Dutt and N.R. Mahapatra, "Parallel A* Algorithms and their Performance on Hypercube Multiprocessors," *Seventh Int'l Par. Proc. Symp.*, pp. 797-803, 1993.

[9] S. Dutt and N.R. Mahapatra, "Scalable Load Balancing Strategies for Parallel A* Algorithms," *Journal of Parallel and Distributed Computing*, Vol.22, No.3, pp.488-505, Sep. 1994.

[10] A. Kavianpour and N. Bagherzadeh, "Application of Three-Valued Logic for Distributed Termination Detection," *Computers and Electrical Engineering*, Vol. 17, No. 2, pp. 65-74, 1991.

[11] D. Kumar, "Development of a Class of Distributed Termination Detection Algorithms," *IEEE Trans. Knowledge and Data Engg.*, Vol. 4, No. 2, Apr. 1992.

[12] V. Kumar, A. Grama, A. Gupta and G. Karypis, "Introduction to Parallel Computing: Design and Analysis of Algorithms," *Benjamin/Cummings Publishing Company*, Redwood City, CA, 1994.

[13] V. Kumar and A. Gupta, "Analyzing scalability of parallel algorithms and architectures," *Journal of Parallel and Distributed Computing*, Vol.22, No.3, pp.379-391, Sep. 1994.

[14] T.-H Lai, Y.-C Tseng and X. Dong, "A More Efficient Message-Optimal Algorithm for Distributed Termination Detection," *Procs. Sixth Int'l Par. Proc. Symp.*, pp. 646-649, Mar. 1992.

[15] N.R. Mahapatra and S. Dutt, "New Anticipatory Load Balancing Strategies for Parallel A* Algorithms," *DIMACS Series in Discrete Mathematics and Theoretical Computer Science—Parallel Processing of Discrete Optimization Problems*, P.M. Pardalos, M.G. Resende, and K.G. Ramakrishnan (eds.), Vol. 22, pp. 197-232, 1995.

[16] F. Mattern, "Algorithm for Distributed Termination Detection," *Distributed Computing*, Vol. 2, pp. 161-175, 1987.

[17] J. Misra, "Detecting termination of distributed computations using markers," *Proc. Second Ann. ACM Symp. on Principles of Distr. Comp.*, pp.290-294, Aug. 1983.

[18] J.M. Rosario and A.N. Choudhary, "High-performance i/o for massively parallel computers: Problems and prospects," *Computer*, pp.59-68, Mar. 1994.

[19] Y. Saad and M.H. Schultz, "Data Communication in Hypercubes," *J. Par. Distr. Comp'g*, Vol.6, pp.115-135, 1989.

[20] P.A. Steenkiste, "A systematic approach to host interface design for high-speed networks," *Computer*, pp.47-58, Mar. 1994.