Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 1047–1066
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

An efficient delay-optimal distributed termination detection algorithm

Nihar R. Mahapatra[a], Shantanu Dutt[b,*,1]

[a]Department of Electrical & Computer Engineering, Michigan State University, East Lansing, MI 48824-1226, USA
[b]Department of Electrical & Computer Engineering, University of Illinois at Chicago, 851 South Morgan Street (M/C 154), Chicago, IL 60607-7053, USA

Received 20 January 1998; received in revised form 23 May 2007; accepted 23 May 2007
Available online 6 June 2007

Abstract

Distributed termination detection is a fundamental problem in parallel and distributed computing and numerous schemes with different performance characteristics have been proposed. 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, on most popular topologies, they take $\Omega(P)$ 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 (which we accurately characterize for an arbitrary topology). On k-ary n-cube tori and meshes, the worst-case delay is $\Theta(D)$, where D is the diameter of the target topology. Further, our algorithm has message and computational complexities of $\Theta(MD + P)$ in the worst case and, for most applications, $\Theta(M + P)$ in the average case—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 first-in first-out (FIFO) ordering for message communication to work correctly.

Keywords: Accumulation; Broadcast; Detection delay; Distributed computation; k-ary n-cubes; Message complexity; Message passing; Termination detection

1. Introduction

1.1. Background

In this paper, we consider efficient algorithms for detecting termination of parallel and distributed computations. The problem of *distributed termination detection* (DTD), as it is often called, is a fundamental one in parallel and distributed computing, with close relationships to other important problems such as deadlock detection, garbage collection, snapshot computation, and global virtual time approximation [36,24]. The model of the computing system used is the same as that in previous work [4] and basically consists of an asynchronous network of P reliable processing elements (PEs) labeled $0, 1, \ldots, P-1$ with diameter D and no shared memory. Each PE is connected to one or more PEs, known as its *neighbors*, by reliable 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 [8]. In such cases, a PE may send messages to or receive messages from any other PE and the messages are assumed to be routed via intermediate PEs or routing switches (in the latter case, no additional computational overhead is incurred by non-neighbor communication) on a shortest path between

* Corresponding author. Fax: +1 312 996 6465
E-mail addresses: nrm@egr.msu.edu (N.R. Mahapatra), dutt@ece.uic.edu (S. Dutt)
URLs: http://www.egr.msu.edu/~nrm (N.R. Mahapatra), http://www.ece.uic.edu/~dutt (S. Dutt).
[1] S. Dutt was supported by NSF Grant MIP-9210049.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.05.013

source and destination PEs. The only effect that the message-passing behavior of the parallel/distributed algorithm has on a DTD algorithm is that the detection delay (to be defined shortly) of the DTD 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 *arbitrary-distance message-passing 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 DTD algorithm is called the *secondary computation* and the messages used by it are termed *secondary messages*. Note that even when the primary computation uses primary messages between only neighboring PEs, the DTD algorithm, depending upon its design, may use secondary messages between non-neighboring PEs; again, the secondary messages are assumed to be routed via intermediate PEs or routing switches on a shortest path between source and destination PEs. As far as the correctness of a DTD algorithm is concerned, message communication latencies may be arbitrary but finite. For the purpose of detection delay analysis of our proposed new DTD algorithm and of related DTD algorithms, however, the following realistic assumptions are made: (1) the time for a message to traverse a single communication link is bounded by a constant; (2) the time to process a simple secondary message that requires fixed processing is bounded by a constant; (3) the number of minimum-payload messages that can be held in the communication buffer of a PE is bounded by a constant; (4) actions required by the secondary computation are given higher precedence by a PE relative to those required by the primary computation (i.e., DTD algorithm actions are not delayed because of primary computation being performed by a PE); and (5) as implicitly assumed in previous work, the effect of communication buffer and link contention on latency per link traversed by a message is bounded by a constant.

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 (or to arbitrary PEs in the arbitrary-distance message-passing case as mentioned previously). (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. 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 [29]. Thus, once a PE becomes idle, it cannot spontaneously become busy. The DTD 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, DTD algorithms need to ensure both these conditions are simultaneously met before signaling termination.

To analyze the performance of DTD algorithms, we will use the following four metrics: (1) *Worst-case detection delay* T_d defined as the worst-case time between the actual completion of the primary computation and its subsequent detection; the worst-case detection delay of DTD algorithms when message passing is between arbitrary PEs will be denoted by T'_d; (2) *Worst-case message complexity* M_s, which is the total number of secondary messages used over all PEs in the worst case; (3) *Worst-case space complexity* S_s measured by the total memory used over all PEs by the secondary computation in the worst case; and (4) *Worst-case computational complexity* C_s over all PEs of the secondary computation.

1.2. Related work

Although DTD is a long-standing problem, new algorithms for it with different performance characteristics and varying assumptions regarding the target computing system model appear regularly [2–4,9,11,14,15,17,18,22,24,25,28,27,29,31,32,36–39]. Some of these algorithms are efficient in terms of the number of secondary messages used [4,17,28], but may take a long time to detect termination [4]. Similarly, some of the algorithms are either less compute intensive [15] or require less memory [14], but are vulnerable to underflow [15] or overflow [14,3] problems.[2] Although the system model described in Section 1.1 is most commonly used, there are algorithms meant for other system characteristics: those that rely on a common clock [9,25,31], or assume a specific target system topology [11], or require first-in first-out (FIFO) communication between PEs [39],[3] or are designed to tolerate PE or link failures [9,18,22,32,37,38]. A general approach to transform a DTD algorithm meant for a fault-free system into that for a system with PE failures (assuming the non-faulty system graph remains connected) is given in [27]. A survey of DTD algorithms appears in [23].

Before proceeding further, it is worth noting that [4] presents a "delayed-DTD" approach that allows their DTD algorithm to work correctly even when started at an arbitrary time after the primary computation has commenced. The advantage of using this approach is that the complexity of the DTD algorithm then depends not on the total number of primary messages M, but rather on the number of primary messages $M' \leqslant M$ sent out after the DTD computation begins. The other advantage is that

[2] In these algorithms, underflow [15] or overflow [14,3] may occur in counters used to keep track of how primary-computation load at a PE gets distributed to other PEs because of dynamic load balancing [15] or the number of messages sent and received by a PE [14,3].

[3] That is, 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.

it enables fault-tolerant DTD: if the system detects the failure of some PEs or links during the primary computation, as long as the system graph remains connected and the integrity of the primary computation itself is not affected (i.e., it provides a meaningful output and eventually terminates), the DTD algorithm can be restarted (after aborting any existing instance of the DTD algorithm) on the residual fault-free system graph [38]. The downside is that the approach relies on FIFO message communication between PEs, requires as many messages as the number of communication links, and introduces nondeterminism into the detection delay (see [21] for explanation). In order to keep the detection delay deterministic so that different DTD algorithms can be compared, and to avoid the restrictive assumption of FIFO capability, we assume in all our performance analyses that the delayed-DTD approach is not used. Therefore, we use M instead of M' in all complexity expressions. Moreover, the delayed-DTD approach is applicable to all DTD algorithms considered in this paper [4]; therefore, the above assumption is a fair one as far as performance comparison of the different algorithms is concerned.

1.3. Shortcomings of related work and our contributions

Existing DTD algorithms meant for general network topologies and system model assumptions described in Section 1.1, e.g. [4,17,28,14], while being efficient with respect to one performance metric, turn out to be inefficient with regard to other metrics. As we will explain shortly, of the four metrics discussed in Section 1.1, detection delay is arguably the most important metric to optimize. In Section 2, we provide a common framework for analyzing DTD algorithm detection delay, which we note depends upon the sum of secondary-message communication time and secondary-message processing time. However, the analysis of detection delays of DTD algorithms in previous work (e.g., in [28]) has commonly ignored two factors critical to accuracy: (a) secondary-message processing time has been ignored and (b) secondary-message communication time between PEs at arbitrary distances has been assumed to be constant. We also explain in Section 2 that termination detection requires collecting state information of all PEs using, say, a tree spanning them. The number of links to be traversed along the longest leaf-to-root path in this tree affects secondary-message communication time, whereas the branching factors of PEs along leaf-to-root paths in this tree influence secondary-message processing time. For any tree embedded in a target system graph, these two parameters are intimately linked: reducing one generally causes the other to increase. In fact, message processing time complexity is equal or higher, in order terms, compared to message communication time complexity on an arbitrary topology in the near-neighbor message-passing case, and so the former should not be ignored in detection delay analysis (as done, e.g., in [28]).

[4] In fact, this observation of ours inspired the formulation of a general approach to transform any DTD algorithm so that it can work correctly when initiated any time after the primary computation has begun [30,29].

Therefore, to correctly analyze the detection delay of a DTD algorithm on a system with a physically realizable topology, both of the previously ignored factors mentioned above must be captured. Our analysis in Section 2 shows that the worst-case detection delay of any DTD algorithm is at least equal in order terms to that for an optimal one-to-all broadcast, which we accurately characterize for an arbitrary topology and show to be $\Theta(D)$ for most popular topologies. In Section 6, we also accurately analyze the worst-case detection delays and other performance metrics of several existing DTD algorithms: three of these use acknowledgments [4,17,28] and one uses message counting to keep track of in-transit primary messages. We find that their worst-case detection delays are $\Omega(P)$ on most system topologies, which is much higher than the optimal complexity of $\Theta(D)$ for these topologies.

Since for most applications the computation and space complexities of the DTD algorithm are modest compared to that of the primary computation, these two complexities are the least important. The message complexity of DTD 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: (1) 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 [33,35], and since detection delay is 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 distributed systems like computational grids (communication times are even longer here) and mobile distributed systems in which message transmission and processing delays are higher because of limited bandwidth, limited processing power, higher error rates, and frequent disconnections [22]. (2) In many applications, results from a primary computation cannot be reliably used before ascertaining its termination, leading to undue delay in their utilization. (3) Sometimes an application consists of many phases, where a new phase can begin only after termination of the previous phase has been detected [6,20]. This requires the use of multiple DTD algorithms one after the other and aggravates both of the above problems.

The rest of the paper is organized as follows. In Section 2, we present a common framework to analyze DTD algorithm detection delay and provide complexity lower bounds for DTD. Next, in Section 3, we present a new DTD algorithm and in Section 4, prove its correctness. We analyze its performance in Section 5 and show that it 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. We also show in Section 5 that our algorithm has optimal or near-optimal values for other performance

metrics as well. We accurately analyze the complexity of several related DTD algorithms and compare them to our new DTD algorithm in Section 6. Finally, we conclude in Section 7.

2. DTD detection delay analysis framework and complexity lower bounds

In this section, after discussing some preliminaries, we present a framework for analyzing the detection delay of and provide lower bounds on the complexity of DTD algorithms. The delay analysis framework will be used to analyze the detection delay of our new DTD algorithm in Section 5.1 and of related DTD algorithms in Section 6. The complexity lower bounds will be used to assess the performance and optimality of our and related DTD algorithms in Section 6.

2.1. Preliminaries

We explain a few concepts and introduce some terms that will prove useful in our discussion and analysis of DTD algorithms, especially those based on acknowledgments (such algorithms are among the most efficient). Acknowledgment-based algorithms employ a spanning *termination tree* with a designated *root PE* and use acknowledgment messages to keep track of in-transit primary messages [4,17,28]. The nonroot vertices of the tree correspond to the other PEs. Note that adjacent tree vertices may not necessarily correspond to neighboring PEs in the target topology. The *distance* $\mathcal{D}(p_i, p_j)$ between any two PEs p_i and p_j is the number of links on the shortest path in the target topology between them. The *length* of a path $\langle p_1, p_2, \ldots, p_m \rangle$ in the termination tree from PE p_1 to PE p_m, where $p_1, p_2, \ldots, p_m$ are all PEs on the termination tree, is defined to be equal to the sum of the distances between consecutive PEs on the path, i.e., it is equal to $\sum_{i=1}^{m-1} \mathcal{D}(p_i, p_{i+1})$. Finally, the *height* of a termination tree is defined to be equal to one less than the number of tree vertices on the root-to-leaf path with the most number of tree vertices on it. Note that the length of the longest root-to-leaf path in a termination tree is greater than its height if there is at least one pair of adjacent tree vertices on the path corresponding to non-neighboring PEs in the target topology; otherwise, the two are equal.

Fig. 1(a) shows a termination tree spanning the PEs of a 5×5 2-D mesh network and illustrates many of the above-defined terms. In it, PE 12 is the root of the termination tree and PEs 0, 1, 2, 3, 4, 5, 11, 13, 16, and 24 are its 10 leaf PEs. Some adjacent vertices on the tree (e.g., PEs 14 and 19) are neighbors in the 2-D mesh, while others (e.g., PEs 19 and 23) are not. The root-to-leaf path with the most number of tree vertices on it is $\langle 12, 17, 22, 21, 20, 15, 16 \rangle$; since there are seven vertices on this path, the height of the termination tree is 6. The root-to-leaf path with the longest length is $\langle 12, 18, 14, 19, 23, 24 \rangle$ and its length is $\mathcal{D}(12, 18) + \mathcal{D}(18, 14) + \mathcal{D}(14, 19) + \mathcal{D}(19, 23) + \mathcal{D}(23, 24) = 2 + 2 + 1 + 2 + 1 = 8$.

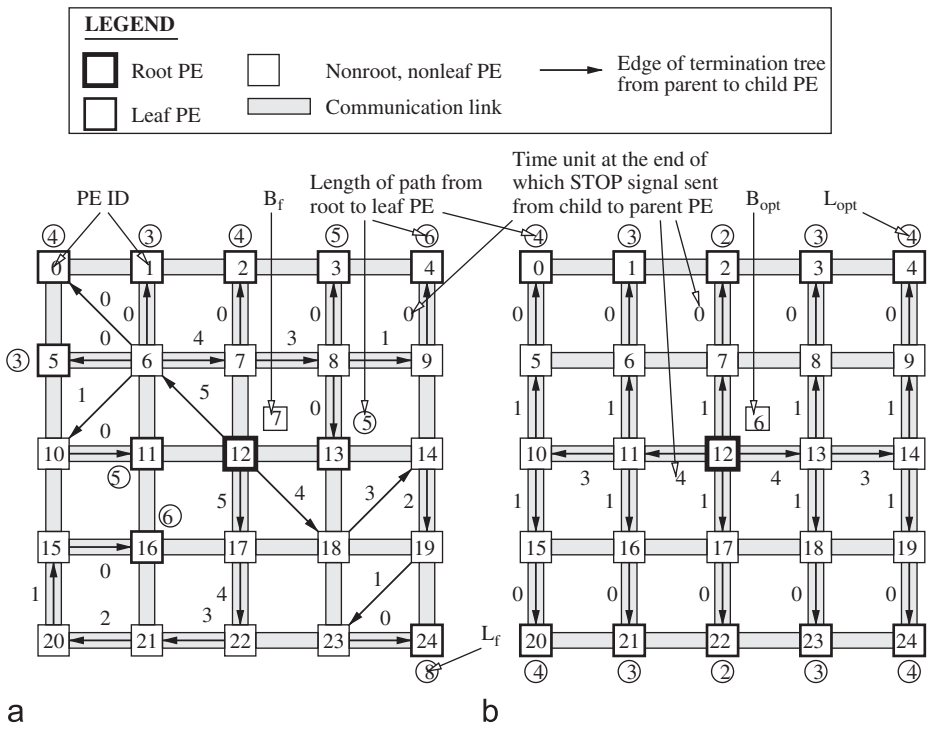

Fig. 1. (a) A tree $\mathcal{T}_f$ spanning the 25 PEs of a 2-D mesh network. The termination tree has height 6 because of root-to-leaf path $\langle 12, 17, 22, 21, 20, 15, 16 \rangle$. The length of the longest root-to-leaf path (viz. $\langle 12, 18, 14, 19, 23, 24 \rangle$) is 8. $\mathcal{L}_{\mathcal{T}_f} = 8$ and $\mathcal{B}_{\mathcal{T}_f} = 7$; note that, during accumulation, STOP messages are sent from a child PE to its parent (i.e., in the reverse direction of arrows shown). (b) One variation of tree $\mathcal{T}_{opt}$ for the 2-D mesh with $L_{opt} = 4$ and $B_{opt} = 6$.

2.2. Common detection delay analysis framework

Next, we briefly discuss some operational characteristics of acknowledgment-based DTD algorithms and introduce a common framework that will be used later to analyze their detection delays. In such algorithms, 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*.

From Definitions 1 and 2, it follows that if a PE is free, it must be idle. Also, a busy PE will be loaded. Moreover, an idle PE will be loaded if at least one primary message sent by it to other PEs has not been acknowledged. Initially, all PEs start out as busy and loaded, and subsequently they may become alternately idle and busy or free and loaded a number of times.

An essential aspect of the termination detection process involves non-root PEs sending a STOP (or similar) message to their parents in the termination tree as soon as they become free and, in the case of non-leaf PEs, have also received STOPs from all their child PEs. Termination is signaled by the root PE when it becomes free and has also received STOPs from all its child PEs. That is, termination involves an all-to-one accumulation in the termination tree, starting at the leaf PEs and ending at the root PE, of the free state of all PEs through the use of STOP messages. When the accumulation is complete, termination can be signaled because it signifies all PEs are free. The latency of this accumulation operation determines the detection delay of DTD algorithms and, as mentioned earlier, depends upon the sum of two factors: secondary-message communication time and secondary-message processing time. Both of these factors depend upon the structure of the termination tree. Our algorithm is designed to use a predetermined static spanning termination tree $\mathcal{T}_{\text{opt}}$ with adjacent vertices on the tree corresponding to neighboring PEs in the target topology and structured so as to minimize the sum of secondary-message communication and processing times on the target topology. Other acknowledgment-based algorithms, on the other hand, employ dynamic termination trees whose structures change, either without [4,17] or within constraints [28], during the primary computation, and so their secondary-message communication and processing times are also different—the root PE, however, always remains the root. Thus, in these other algorithms, although the termination tree may initially start as $\mathcal{T}_{\text{opt}}$ when the primary-computation commences, the termination tree $\mathcal{T}_{\text{f}}$ at the time of termination, unlike in our algorithm, may be quite different, with edges that are not in $\mathcal{T}_{\text{opt}}$.

We next accurately characterize the detection delay of acknowledgment-based DTD algorithms by analyzing the time complexity of the above-described accumulation operation during DTD. This can be done by analyzing secondary-message communication and processing times separately. First, if message processing time is ignored, accumulation delay depends soley upon secondary-message communication time, which is clearly proportional to the length $\mathcal{L}_{\mathcal{T}_{\text{f}}}$ of the longest root-to-leaf path in $\mathcal{T}_{\text{f}}$. That is, $\Theta(\mathcal{L}_{\mathcal{T}_{\text{f}}})$ is the zero-processing-delay secondary-message communication time. For example, for the tree $\mathcal{T}_{\text{f}}$ in Fig. 1(a), $\mathcal{L}_{\mathcal{T}_{\text{f}}} = 8$—note that the effect of buffer and link contention during communication is ignored in this analysis as stated and for reasons given in Section 1.1.

Next, if message communication time is ignored, accumulation delay depends soley upon secondary-message processing time. In this case, the time $\mathcal{B}_{\mathcal{T}_{\text{f}}}$ at which the accumulation operation completes at the root PE can be computed recursively starting from leaf PEs and proceeding toward the root PE as follows. Assume the accumulation operation starts at time 0 with leaf PEs in $\mathcal{T}_{\text{f}}$ sending to their parent PEs a STOP message, each of which takes unit time to process. A PE sends a STOP to its parent PE in $\mathcal{T}_{\text{f}}$ as soon as it has received and processed STOPs from all its child PEs (and it is itself free)—note that the STOP is instantaneously received by the parent PE because we are ignoring message communication time in this analysis. Suppose a PE p_i in $\mathcal{T}_{\text{f}}$ receives STOPs simultaneously from $n_{p_i,l}$ child PEs at time $t_{p_i,l}$, where $n_{p_i,l} \geqslant 1$, $1 \leqslant l \leqslant m_{p_i}$ (i.e., m_{p_i} is the number of distinct times STOP messages arrive at p_i from its child PEs during the accumulation operation), and $0 \leqslant t_{p_i,1} < t_{p_i,2} < \cdots < t_{p_i,l-1} < t_{p_i,l}$. The time at which the first set of STOPs received at time $t_{p_i,1}$ would have been processed is $b_{p_i,1} = t_{p_i,1} + n_{p_i,1}$, the time at which the second set of STOPs received at time $t_{p_i,2}$ would have been processed is $b_{p_i,2} = \max(t_{p_i,1} + n_{p_i,1} + n_{p_i,2}, t_{p_i,2} + n_{p_i,2})$, and so on. Therefore, the time at which all STOPs would have been processed by p_i and the time at which a STOP will be sent to the parent PE is $b_{p_i,m_{p_i}} = \max(t_{p_i,m_{p_i}-1} + n_{p_i,m_{p_i}-1} + n_{p_i,m_{p_i}}, t_{p_i,m_{p_i}} + n_{p_i,m_{p_i}})$; if p_i is the root PE, instead of sending a STOP, it will signal termination at time $b_{p_i,m_{p_i}} = \mathcal{B}_{\mathcal{T}_{\text{f}}}$. In this recursive manner, the zero-communication-delay secondary-message processing time $\Theta(\mathcal{B}_{\mathcal{T}_{\text{f}}})$ can be computed. For example, for the tree $\mathcal{T}_{\text{f}}$ in Fig. 1(a), $\mathcal{B}_{\mathcal{T}_{\text{f}}} = 7$.

Both $\mathcal{L}_{\mathcal{T}_{\text{f}}}$ and $\mathcal{B}_{\mathcal{T}_{\text{f}}}$ depend upon the structure of $\mathcal{T}_{\text{f}}$. Clearly, when both secondary-message communication and processing times are considered, accumulation delay is at least as much as the larger of the two (i.e., it is $\Omega(\max(\mathcal{L}_{\mathcal{T}_{\text{f}}}, \mathcal{B}_{\mathcal{T}_{\text{f}}})) = \Omega(\mathcal{L}_{\mathcal{T}_{\text{f}}} + \mathcal{B}_{\mathcal{T}_{\text{f}}})$) and, since message communication and processing may overlap during accumulation, it is no more than the sum of the two (i.e., it is $O(\mathcal{L}_{\mathcal{T}_{\text{f}}} + \mathcal{B}_{\mathcal{T}_{\text{f}}})$). This leads to the following lemma.

Lemma 1. *For a termination tree $\mathcal{T}_{\text{f}}$ at the time of termination in an acknowledgment-based DTD algorithm, the time complexity of an all-to-one accumulation of the free states of PEs in $\mathcal{T}_{\text{f}}$ is $\Theta(\mathcal{L}_{\mathcal{T}_{\text{f}}} + \mathcal{B}_{\mathcal{T}_{\text{f}}})$, where $\Theta(\mathcal{L}_{\mathcal{T}_{\text{f}}})$ is the zero-processing-delay secondary-message communication time and $\Theta(\mathcal{B}_{\mathcal{T}_{\text{f}}})$ is the zero-communication-delay secondary-message processing time.*

Consequently, in this and later sections, we will analyze DTD detection delay complexity by analyzing each of secondary-message communication and processing times separately while ignoring the other.

We next define a few terms related to secondary-message communication and processing times that will be useful in

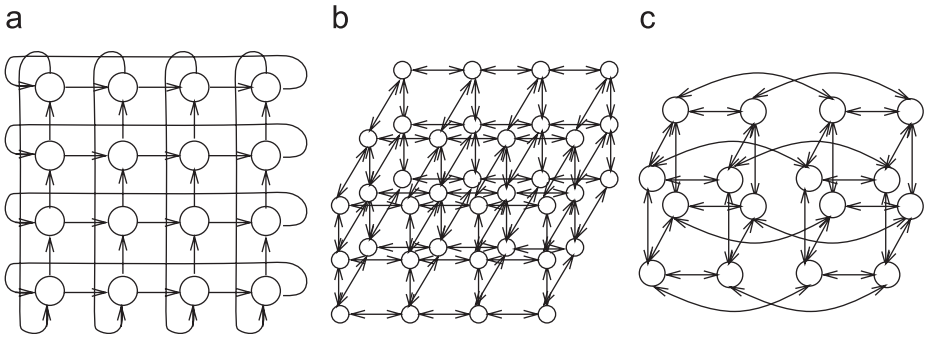

Fig. 2. Examples of k-ary n-cube tori/meshes: (a) 2-D torus ($n = 2$), (b) 3-D mesh ($n = 3$), and (c) hypercube ($n = \log_2 P$).

characterizing the detection delay complexity of DTD algorithms later. Let $L_{\max,h}$ ($L'_{\max,h}$) denote the maximum possible value for the length of a root-to-leaf path in a termination tree of height h embedded in the target topology, such that vertices that are adjacent on the tree are also (may not necessarily be) neighbors in the target topology. Define $B_{\max}$ ($B'_{\max}$) to be equal to $\mathcal{B}_{\mathcal{T}}$ for a termination tree $\mathcal{T}$ embedded in the target topology and structured so as to maximize it, such that vertices that are adjacent on the tree are also (may not necessarily be) neighbors in the target topology. Finally, let $L_{\text{opt}} \equiv \mathcal{L}_{\mathcal{T}_{\text{opt}}}$ and $B_{\text{opt}} \equiv \mathcal{B}_{\mathcal{T}_{\text{opt}}}$. Recall that $\mathcal{T}_{\text{opt}}$ in our algorithm is structured so as to optimize its worst-case detection delay, which is $\Theta(L_{\text{opt}} + B_{\text{opt}})$. For example, for the 2-D mesh of Fig. 1(a), (one possible variation of) $\mathcal{T}_{\text{opt}}$ is given in Fig. 1(b); for this tree, $L_{\text{opt}} = 4$ and $B_{\text{opt}} = 6$.

The following lemma provides bounds on the above-defined termination tree properties for arbitrary topologies.

Lemma 2. *For termination trees used in acknowledgment-based DTD algorithms on an arbitrary target topology of diameter D and containing P PEs*: (1) $L_{\max,h} = h$; (2) $L'_{\max,h} = \Omega(h)$, $L'_{\max,h} = O(hD)$, $L'_{\max,D} = \Theta(D^2)$, $L'_{\max,P-1} = \Omega(D^2 + P)$, *and* $L'_{\max,P-1} = O(PD)$; (3) $L_{\text{opt}} = \Theta(D)$; (4) $B_{\max} = \Omega(D)$ and $B_{\max} = O(P)$; (5) $B'_{\max} = P - 1$; *and* (6) $B_{\text{opt}} \geqslant L_{\text{opt}}$, $B_{\text{opt}} = \Omega(D)$, *and also* $B_{\text{opt}} = O(\min(Dd_{\max}, P))$, *where $d_{\max}$ is the maximum degree of a PE in the target topology.*

Proof sketch. Most of the results are self-evident and require no explanation. $L'_{\max,h} = O(hD)$ since consecutive PEs on a root-to-leaf path may be $O(D)$ distance apart. $L'_{\max,D} = \Theta(D^2)$ because if we consider a path $\langle p_0, p_1, p_2, \ldots, p_D\rangle$ in which consecutive PEs p_i and p_{i+1}, $0 \leqslant i < D$, are neighbors in the target topology, then the ordering $\langle p_0, p_{\lceil D/2\rceil}, p_1, p_{\lceil D/2\rceil+1}, p_2, p_{\lceil D/2\rceil+2}, \ldots\rangle$ is one in which consecutive PEs are $\Theta(D)$ distance apart. Similarly, $L'_{\max,P-1} = \Omega(D^2 + P)$ because a chain of $\Theta(D)$ PEs can have a length of $\Theta(D^2)$ and the remaining $\Theta(P)$ PEs can be ordered to have a length of at least $\Theta(P)$. $B'_{\max} = P - 1$ since the termination tree may have a star structure. □

To derive tight bounds for $B_{\max}$, $L'_{\max,h}$, and B_{opt}, and hence for worst-case detection delays of various algorithms, we specifically consider an interesting and very general class of topologies called k-ary n-cube tori, which includes rings (n=1), 2-D tori (n=2), 3-D tori (n=3), and hypercubes (k=2) [8] (see Fig. 2). Here, n is referred to as the *dimension* and k the *radix* of the topology. Every PE 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 number of PEs in a k-ary n-cube is $P = k^n$ and its diameter is $n\lfloor k/2\rfloor$. Another class of topologies called k-ary n-cube meshes is also very general, and differs from the k-ary n-cube tori class only in that its members do not have end-around connections in any dimension; the linear array ($n = 1$), 2-D mesh ($n = 2$), and 3-D mesh ($n = 3$) are its special cases (see Fig. 2).

Tight bounds for $B_{\max}$, $L'_{\max,h}$, and B_{opt} for k-ary n-cubes are provided in the next lemma.

Lemma 3. *For termination trees used in acknowledgment-based DTD algorithms on an arbitrary k-ary n-cube torus or mesh of diameter D and containing P PEs*: (1) $B_{\max} = \Theta(P)$; (2) $L'_{\max,h} = \Theta(hD)$; *and* (3) $B_{\text{opt}} = \Theta(D)$.

Proof sketch. First, $B_{\max} = \Theta(P)$ because a linear ordering $G(k, n) = \langle p_0, p_1, \ldots, p_{k^n-1}\rangle$ of the P PEs in which consecutive PEs are neighbors can be obtained using a reflected Gray code, where $p_i \equiv a_{i,n-1}a_{i,n-2}\ldots a_{i,1}a_{i,0}$, $0 \leqslant i < P$, denotes the label of the ith PE in the ordering. We prove the second result by deriving another linear ordering $G'(k, n)$ of the P PEs in which consecutive PEs are $\Theta(D)$ distance apart by swapping suitable pairs of PEs in $G(k, n)$. For example, $G(6, 2) = \langle 00, \underline{01}, 02, \underline{03}, 04, \underline{05}, 15, \underline{14}, 13, \underline{12}, 11, \underline{10}, 20, \underline{21}, 22, \underline{23}, 24, \underline{25}, 35, \underline{34}, 33, \underline{32}, 31, \underline{30}, 40, \underline{41}, 42, \underline{43}, 44, \underline{45}, 55, \underline{54}, 53, \underline{52}, 51, \underline{50}\rangle$ and $G(5, 2) = \langle 00, \underline{01}, 02, \underline{03}, 04, \underline{14}, 13, \underline{\overline{12}}, 11, \underline{10}, 20, \underline{\overline{21}}, 22, \underline{\overline{23}}, 24, \underline{34}, 33, \underline{\overline{32}}, 31, \underline{30}, 40, \underline{41}, 42, \underline{43}, 44\rangle$, where "swappable" PEs—to be defined shortly—are underlined and PEs at odd-numbered positions that are not swappable are overlined. Note that when k is even, P is even, when k is odd, P is odd, and that there are an even number of PEs at odd-numbered positions (i.e., p_i such that i is odd), regardless of whether k is even or odd. Let $a'_{i,j} = a_{i,j} + \lceil k/2\rceil \bmod 2\lceil k/2\rceil$. For all PEs p_i such that i is odd and $a'_{i,j} < k$, $0 \leqslant j < n$—these are referred to

as *swappable PEs*—let $p_l = \mathcal{F}(p_i) \equiv a'_{i,n-1}a'_{i,n-2}\ldots a'_{i,1}a'_{i,0}$. It can be shown that: (1) l is odd and p_l is swappable; (2) $\mathcal{F}(\mathcal{F}(p_i)) = p_i$ or $\mathcal{F} = \mathcal{F}^{-1}$; (3) all $P/2 = k^n/2$ PEs at odd-numbered positions in $G(k, n)$, when k is even, are swappable; (4) when k is odd, only those PEs p_i at odd-numbered positions for which any $a_{i,j} = \lfloor k/2 \rfloor$, $0 \leqslant j < n$, are not swappable—the number of such PEs is clearly $\lfloor \frac{k^n-(k-1)^n}{2} \rfloor$ or the number of swappable PEs is the even number $(k-1)^n/2 = \Theta(k^n) = \Theta(P)$. For example, in $G(6, 2)$, $p_{15} = 23$, $p_{35} = 50$, $\mathcal{F}(p_{15}) = p_{35}$, and $\mathcal{F}(p_{35}) = p_{15}$, and in $G(5, 2)$, $p_7 = 12$ is a PE at an odd-numbered position that is not swappable, $p_5 = 14 = \mathcal{F}(p_{20}) = \mathcal{F}(41)$. Let $G'(k, n)$ denote a linear chain of PEs obtained from $G(k, n)$ by replacing all swappable PEs p_i in the latter by $\mathcal{F}(p_i)$. For example, $G'(6, 2) = \langle 00, \underline{34}, 02, \underline{30}, 04, \underline{32}, 15, \underline{41}, 13, \underline{45}, 11, \underline{43}, 20, \underline{54}, 22, \underline{50}, 24, \underline{52}, 35, \underline{01}, 33, \underline{05}, 31, \underline{03}, 40, \underline{14}, 42, \underline{10}, 44, \underline{12}, 55, \underline{21}, \underline{53}, \underline{25}, 51, \underline{23} \rangle$ and $G'(5, 2) = \langle 00, \underline{34}, 02, \underline{30}, 04, \underline{41}, 13, \overline{\underline{12}}, 11, \underline{43}, 20, \overline{\underline{21}}, 22, \overline{\underline{23}}, 24, \underline{01}, 33, \overline{\underline{32}}, 31, \underline{03}, 40, \underline{14}, 42, \underline{10}, 44 \rangle$. Since a termination tree of height h, $0 \leqslant h < P$, can correspond to a subchain consisting of $h + 1$ consecutive PEs in $G'(k, n)$, $L'_{\max,h} = \Theta(hD)$.

Finally, for the third result, we consider an optimal one-to-all broadcast in a k-ary n-cube torus, 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.[5] There are in all n stages, with $\Theta(k/2)$ steps per stage. In each stage i, for $1 \leqslant i \leqslant n$, all PEs with labels of 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 $0^{n-i}bx$—the broadcast is accomplished in $\Theta(k/2)$ steps by first sending the data from PE $0^{n-i+1}x$ to its two neighbors on the ring (one neighbor in the case of a hypercube for which $k = 2$) and then spreading it in opposite directions on the ring; here, x is any $(i-1)$-digit radix-k label and $1 \leqslant b < k$. This broadcast tree can also be used for an all-to-one accumulate and for DTD by reversing the direction of communication in it, and is, in fact, the tree $\mathcal{T}_{\text{opt}}$ used in our DTD algorithm. During the accumulate, all communication is between neighboring PEs only and any PE at a given time receives and processes only a fixed number of messages (0, 1, or 2) from its child PEs. Since there are n stages in all, with $\Theta(k/2)$ steps per stage, L_{opt} and B_{opt} both $= \Theta(nk/2) = \Theta(D)$. For example, an optimal broadcast/accumulate tree for a 5-ary 2-cube mesh (i.e., a 2-D mesh) with $L_{\text{opt}} = 4$ and $B_{\text{opt}} = 6$ is shown in Fig. 1(b).

Similar results can be established for k-ary n-cube meshes. □

2.3. Complexity lower bounds for DTD algorithms

Note that in any DTD algorithm, in the worst case, information about the states of all PEs needs to be collected for DTD after termination has occurred. This essentially requires an all-to-one accumulation operation at some PE to gather the information. An optimal all-to-one accumulation operation at, say, the root PE can be performed using the termination tree $\mathcal{T}_{\text{opt}}$ referred to earlier since it minimizes $\Theta(L_{\text{opt}} + B_{\text{opt}})$, which captures the worst-case message communication and processing delays for the accumulation operation. An optimal one-to-all broadcast has the same complexity as an optimal all-to-one accumulation operation and can be performed by reversing the communication paths in the latter [34]. It is shown in [7,4] that M_s of any DTD algorithm is $\Omega(M + P)$. Since each secondary message is associated with some computational overhead, C_s of any DTD algorithm also has the same complexity. 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 $\Omega(P)$. Therefore, from Lemmas 2 and 3, we obtain:

Theorem 4. *Any DTD algorithm, on an arbitrary target topology of diameter D and containing P PEs, has a worst-case detection delay that is at least equal in order terms to the time for an optimal all-to-one accumulate or one-to-all broadcast on it. Specifically, for any DTD algorithm,* $T_d = \Omega(B_{\text{opt}})$, $T'_d = \Omega(B_{\text{opt}})$, $M_s = \Omega(M + P)$, $S_s = \Omega(P)$, *and* $C_s = \Omega(M + P)$, *where M is the number of primary messages used by the primary computation. On k-ary n-cubes,* $T_d = \Omega(D)$ *and* $T'_d = \Omega(D)$.

3. Our algorithm: STATIC_TREE_DTD

3.1. Basic idea

We now present our new DTD algorithm which, like the algorithms in [4,17], uses a spanning termination tree for DTD. However, as explained in Section 2.2, our termination tree $\mathcal{T}_{\text{opt}}$ is static: it is rooted at a *root PE*, with adjacent vertices on the tree corresponding to neighboring PEs in the target topology, and is structured so as to optimize a one-to-all broadcast from the root PE (i.e., to optimize $\Theta(L_{\text{opt}} + B_{\text{opt}})$).[6] Therefore, we refer to our algorithm as STATIC_TREE_DTD. 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 parent PE when they have finished their primary computation. Since the worst-case secondary-message communication and processing delay is $\Theta(L_{\text{opt}} + B_{\text{opt}})$, the worst-case branching-in process time, and hence the worst-case detection delay of our algorithm, as will be shown later, is also on the same order.

The key to our algorithm's reduced detection delay is that it is structured so that we are consistently able to regard the

[5] Superscripts in labels denote concatenation.

[6] 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 [34]. 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 [10].

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 [4,17] 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 [4] (see Section 6.1), 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 acknowledgment to the preceding PE and it can only do so after receiving one from the next PE in the chain. In [17] (see Section 6.1), 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 $\Theta(P)$ acknowledgment 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.

3.2. 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 Definitions 1 and 2; the last two will be defined a little later. Note from Definitions 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 DTD 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 PEs 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 (excluding PE j) 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 non-root 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 received from one of its children. 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 our STATIC_TREE_DTD algorithm is given in Fig. 3.

3.3. Algorithm illustration

In Fig. 4, we show a spanning tree mapped onto some target topology to illustrate the above DTD 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

Algorithm STATIC_TREE_DTD(i)
/* Detects termination of a parallel primary computation on P PEs */
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 $idle = 1$ (0) $\Rightarrow$ PE i idle (busy); $free = 1$ (0) $\Rightarrow$ PE i free (loaded); $inactive = 1$ (0) $\Rightarrow$ PE i inactive (active); $child\_inactive[j]$ = (# STOP messages) − (# RESUME messages) received from child j of i; num_unack_msgs = # unacknowledged primary messages sent out by i; $terminated = 1$ (0) $\Rightarrow$ PE i has (has not) detected termination. The terms parent and child are used in reference to $\mathcal{T}_{\text{opt}}$. */

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) $child\_inactive[j] + +$.
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 $M_{i,j}$) $num\_unack\_msgs + +$.
   /* $M_{i,j}$ denotes a message from PE i to 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**
   $child\_inactive[l] - -$;
   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** $num\_unack\_msgs - -$;
   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 STATIC_TREE_DTD */

Fig. 3. Pseudocode for algorithm STATIC_TREE_DTD.

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. 4(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. 4(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. 4(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. 4(d)).

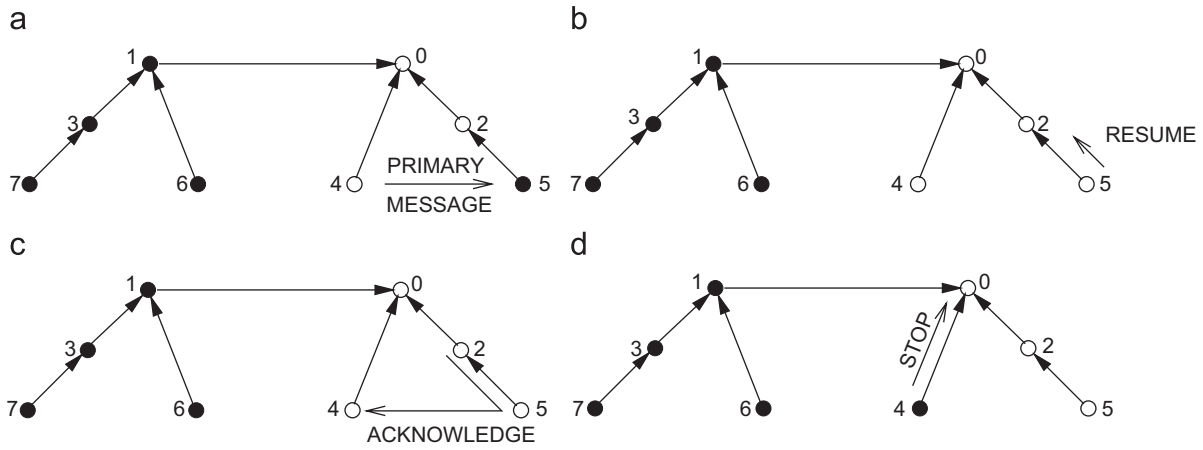

Fig. 4. Illustration of STATIC_TREE_DTD procedure on an arbitrary topology. (a) PE 4 sends a primary message to PE 5. (b) PE 5 resumes and sends a RESUME message up the termination tree to PE 2. (c) PE 2 sends an ACKNOWLEDGE message to PE 4 via PE 5. (d) PE 4 relinquishes ownership of the previously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

4. Proof of correctness of STATIC_TREE_DTD

4.1. Preliminaries

Before proving the correctness of algorithm STATIC_TREE_DTD, we need to define a few terms. All algorithm statements cited hereafter refer to the statements in STATIC_TREE_DTD. 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 $t \geqslant t_0$, a PE is either active or inactive. By Definition 3, the root PE is active until it signals termination.

Definition 4. The *state* of the termination tree at any time $t \geqslant t_0$ 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*, which are defined as follows.

Definition 5. A *STOP event* is said to *start* at an active non-root PE u when it becomes inactive and reports a STOP to its parent, or in the case of the root PE, when it reports TERMINATION. This is followed by a sequence of STOPs from child to parent, one after another, along the path from u toward the root PE. The STOP event is said to be *complete* when the STOP from u either: (1) reaches an ancestor PE that is not free or is waiting for at least one other STOP, or when (2) it reaches the root PE and the root PE signals TERMINATION. The *STOP path* corresponding to u for this event is the set of PEs, ordered from u toward the root PE, that report STOPs (or signal TERMINATION in the case of the root PE).

In the above, PEs on the STOP path become inactive.

Definition 6. A *RESUME event* at an inactive PE v is said to *start* when it receives a primary message, becomes active, and reports a RESUME to its parent. This is followed by a sequence of RESUMEs from child to parent, one after another, along the path from v toward the root PE, until the RESUME reaches an active PE, and the RESUME event *completes*. The *RESUME path* corresponding to v for this event is the set of PEs, ordered from v toward the root PE, that report RESUMEs.

In the above, PEs on the RESUME path become active.

Definition 7. By a *state event* we mean either a STOP or a RESUME event. Two state events are said to be *simultaneous* if the time periods of their occurrence from start to completion have any overlap. A set of state events is considered to be *simultaneous* if for every pair of events (i, j) in the set, there exists a sequence of events $\langle i, e_1, e_2, \ldots, e_m, j\rangle$ in the set, where $m \geqslant 0$, such that any pair of consecutive events in the sequence are simultaneous.

With the above preliminaries out of the way, we next prove that our algorithm does not incorrectly signal termination.

4.2. Proof of no incorrect termination detection

First, in Lemma 5, we relate the effect of a set of simultaneous state events to the net effect of the state events occurring in a particular sequence, without any overlap, assuming 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 6 that STATIC_TREE_DTD does not incorrectly signal termination when FIFO communication takes place. Finally, in Theorem 7 we show that using increment and decrement operations on the elements of *child_inactive* correctly handles the non-FIFO case as well.

4.2.1. FIFO case

Lemma 5 (*Mahapatra and Dutt [21]*). *Assume that message communication in links is in FIFO order and that the elements of child_inactive in statements* 4 *and* 8 *of* STATIC_TREE_DTD

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*.

We refer the reader to [21] for a proof to the above lemma.

Theorem 6. *Assuming message communication in links is in FIFO order and that in statements* 4 *and* 8 *of* STATIC_TREE_DTD *the elements of child_inactive are set to* 1 *and reset to* 0, *respectively*, STATIC_TREE_DTD *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 5 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. We already 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 Definition 5, the only active child of a PE on the STOP path is also on the 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 $\mathcal{T}_m$ 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 Definition 6, the parent of the last inactive PE on the RESUME path is active and hence 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. Figs. 5(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 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 acknowledgments). 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, STATIC_TREE_DTD will not signal termination when there is primary computation to be performed, thus proving Theorem 6. □

4.2.2. Non-FIFO case

Theorem 7. STATIC_TREE_DTD *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, $0 \leqslant$ (number of STOPs sent out by j) $-$ (number of RESUMEs sent out by j) $\leqslant 1$. 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.

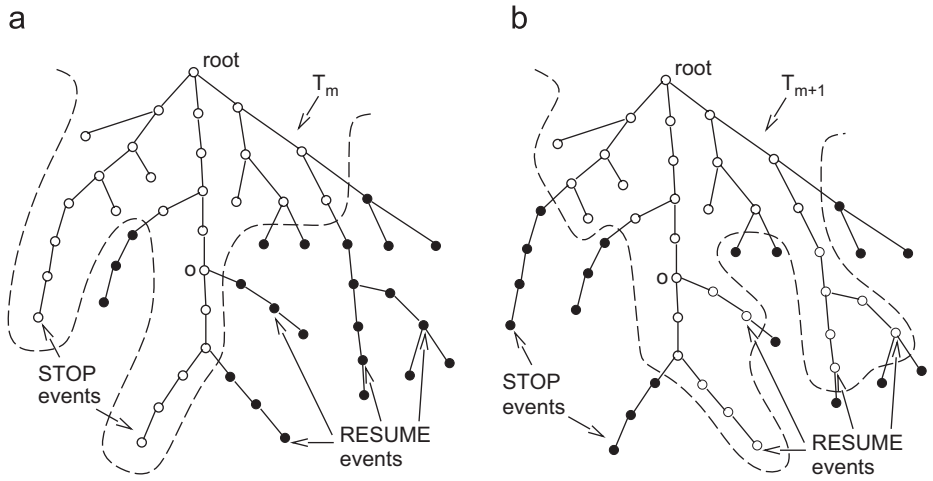

Fig. 5. (a) State of the termination tree before the $(m+1)$th state event starts showing the active tree $\mathcal{T}_m$ at that time, and (b) state of the termination tree after the $(m+1)$th state event completes showing the new active tree $\mathcal{T}_{m+1}$. White circles denote active PEs and black ones inactive PEs. The dashed curve demarcates the active tree.

However, this will not result in signaling of termination since the sender PEs corresponding to the pending RESUME messages (and hence pending acknowledgments) 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 elements of 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 6. □

4.3. 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 Section 5.

Theorem 8. STATIC_TREE_DTD *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 acknowledgments have 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 toward 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 acknowledgments (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 toward the root PE. Since the termination tree has a finite height, the root PE will eventually become inactive a finite time after t_a and signal termination. □

Theorem 9. STATIC_TREE_DTD *correctly detects termination of any parallel primary computation, irrespective of whether messages in links are communicated in FIFO order or not.*

Proof. From Theorems 6 and 7, STATIC_TREE_DTD will not signal termination incorrectly if the primary computation is not yet completed. The rest of the proof follows from Theorem 8. □

5. Complexity analysis of STATIC_TREE_DTD

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

5.1. 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 [10]. 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 at height 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.

Next, we establish the best- and worst-case detection delays of our algorithm.

Theorem 10. STATIC_TREE_DTD, *on an arbitrary target topology of diameter D and containing P PEs, has a best-case detection delay of* $\Theta(1)$ *and finite optimal worst-case detection delays on the same order as an optimal one-to-all broadcast* (*or all-to-one accumulation*) *on the target topology of* T_d *and* T'_d *both* $= \Theta(B_{opt})$, *where* $B_{opt} = \Omega(D)$ *and also* $B_{opt} = O(\min(Dd_{max}, P))$ *and* d_{max} *is the maximum degree of a PE in the target topology.*

Proof. After the primary computation ends and termination occurs at time t_a, RESUME and ACKNOWLEDGE messages corresponding to some yet unacknowledged primary messages may be passed up and down the termination tree, respectively, till a later time t'_a (see statements 7–9 in Fig. 3). It may take till an even later time t''_a by when any pending ACKNOWLEDGE messages from recipient PEs of primary messages have been received and processed by all sender PEs. From the proof for Theorem 8, after t''_a, the active tree $\mathcal{T}_a$ at time t''_a can only contract toward 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 acknowledgments. 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 toward 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 first show that the active-tree contraction process between t''_a and t'''_a in the worst case is similar to an all-to-one accumulation operation at the root PE, where the messages involved are STOP messages. Then, we will show that the time intervals $t'_a - t_a$ and $t''_a - t'_a$ are $O(t'''_a - t''_a)$ in the worst case. Thus, $T_d = T'_d = \Theta(t'''_a - t''_a)$.

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 at height 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 $t'_b = t_b + n_{p_i,1}\tau_m = t''_a + \tau_c + n_{p_i,1}\tau_m$, so that the active-tree contracts by one level toward the root PE; here, $n_{p_i,1}$ is the number of children that a PE p_i (which is one of the new leaf PEs at height 1 that became inactive) has on the termination tree $\mathcal{T}_{opt}$ and is equal to the number of STOPs received (and processed) by it at time t_b (see Section 2.2), 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. Clearly, as the active tree continues to contract, additional time for secondary-message communication and processing is spent (see Section 2.2 for all-to-one accumulation delay). Therefore, the total active-tree contraction time $t'''_a - t''_a = L_{opt}\tau_c + B_{opt}\tau_m$. From Lemma 2, $L_{opt} = \Theta(D)$, $B_{opt} \geqslant L_{opt}$, $B_{opt} = \Omega(D)$, and also $B_{opt} = O(\min(Dd_{max}, P))$.

Second, consider the time $t'_a - t_a$ in the worst case. 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 (along with the root PE). These RESUME messages will reach PEs at height 1 in the termination tree, each of which will become active upon receiving and processing the first RESUME. This first RESUME will be passed up the termination tree to the parent PE soon after it is processed, while each of the RESUMEs received later (when the PE is already active) will cause an ACKNOWLEDGE message to be sent in the opposite direction of the corresponding RESUME soon after previous RESUMEs and it are processed. Thus, RESUME messages from leaf PEs will be passed up the termination tree toward the root PE to different extents along their leaf-to-root paths and will incur different processing delays at the PEs they pass through (depending upon how many RESUMEs are received earlier). ACKNOWLEDGE messages passed down the termination tree incur constant processing delay at each of the PEs they pass through. Clearly, there is no contention between different RESUME and ACKNOWLEDGE messages and the total secondary-message communication and processing delays incurred by any RESUME or ACKNOWLEDGE is no more than that incurred during an all-to-one accumulate or one-to-all broadcast. Therefore, from the time RESUME messages were sent by leaf PEs, all ACKNOWLEDGE messages will have reached their corresponding leaf PEs in $t'_a - t_a = O(B_{opt})$ time.

Next, consider the time interval from t'_a to t''_a in the worst case. ACKNOWLEDGE messages will be sent from the recipient PEs of primary messages to the corresponding sender PEs. Consider one such sender PE. Clearly, after a primary message is issued by the sender PE, it will take $t_1 = \Theta(D)$ time in the worst case to reach the recipient PE. Once received, the primary message will be processed by the recipient PE within $t_2 = \Theta(1)$ time after processing up to $\Theta(1)$ messages received earlier and waiting to be processed in the communication buffer of the recipient PE (see Section 1.1). After the primary message is processed by the recipient PE, it will take a further $t_3 = O(B_{opt})$ time in the worst case (during which RESUME and ACKNOWLEDGE messages are passed up and down the termination tree, respectively, as discussed in the previous paragraph) before an ACKNOWLEDGE message is sent by the recipient to the sender PE, where it will reach in another $t_4 = \Theta(D)$ time and be processed in an additional $t_5 = \Theta(1)$ time in the worst case after processing up to $\Theta(1)$ messages received earlier and waiting to be processed in the communication buffer of the sender PE. Since $B_{opt} = \Omega(D)$ from Lemma 2, the total time elapsed from the time a primary message is sent and the corresponding ACKNOWLEDGE is received and processed by

a sender PE in the worst case is $t_1+t_2+t_3+t_4+t_5 = O(B_{\text{opt}})$. In other words, in the worst case, this elapsed time is cB_{opt} or less, where c is some constant.

The time interval $t''_a - t'_a$ will be maximum when a sender PE has the most number of unacknowledged primary messages at the time of termination, ACKNOWLEDGE messages for which it will then have to process after termination. This will occur when the sender PE: (a) issues a primary message every time unit, except during a time unit when it processes a received secondary message—recall from Section 1.1 that secondary computation actions (such as processing of a received secondary message) take precedence over primary computation actions (such as issuing of a primary message); and (b) the elapsed time between the issuing of a primary message to the receipt and processing of the corresponding ACKNOWLEDGE is maximum, which, for the purpose of a worst-case analysis, can be assumed to be cB_{opt} (although it may be less). If we consider such a sender PE from the beginning of primary computation at time 0, it will issue a primary message every time unit from time unit 1 to time unit cB_{opt}. From time unit $cB_{\text{opt}}+1$ to time unit $2cB_{\text{opt}}$, it will be busy processing the ACKNOWLEDGE messages corresponding to the primary messages sent earlier as they are received one after another. Again, from time unit $2cB_{\text{opt}}+1$ to time unit $3cB_{\text{opt}}$, it will issue primary messages every time unit and from time unit $3cB_{\text{opt}}+1$ to time unit $4cB_{\text{opt}}$, it will process the ACKNOWLEDGE messages corresponding to them, and so on. Therefore, at any time (including at the time of termination), the maximum number of primary messages yet to be acknowledged is $O(B_{\text{opt}})$. Since it takes $\Theta(D)$ time in the worst case for ACKNOWLEDGE messages to reach sender PEs from recipient PEs after time t'_a and only $O(B_{\text{opt}})$ ACKNOWLEDGE messages are processed after termination, $t''_a - t'_a = O(B_{\text{opt}})$.

Hence, T_{d} and T'_{d} both $= \Theta(B_{\text{opt}})$. From Theorem 4, this is also optimal and is the same complexity as that for an optimal all-to-one accumulate or one-to-all broadcast on the same topology. □

Since from Lemma 3 $B_{\text{opt}} = \Theta(D)$, we obtain the following result.

Corollary 11. STATIC_TREE_DTD *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 topology.*

5.2. Message complexity

We now consider the message complexity of STATIC_TREE_DTD. Since each primary message can potentially cause $\Theta(D)$ RESUME messages before an acknowledgment for it is issued, the message complexity can be as much as $\Theta(MD+P)$. However, on the average, the message complexity will likely be $\Theta(M+P)$ because a single primary message is likely to cause multiple RESUMEs only toward 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.

5.2.1. Average message complexity

Theorem 12. *Assuming that all PEs have the same free probability (i.e., the probability of being free)* $q(t)$ *at any time* t *in the interval* (t_0, t_a), *and that* $\frac{1}{1-q(t)}$ *(i.e., the reciprocal of the loaded probability) averaged over* (t_0, t_a) *is bounded above by a constant, the average message complexity* $M_{\text{s,avg}}$ *of* STATIC_TREE_DTD *is* $\Theta(M+P)$ *on all topologies. 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 STATIC_TREE_DTD are STOP, RESUME, ACKNOWLEDGE, and TERMINATION. The number of STOPs used is at least $\Theta(P)$ since every PE, except root, 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 STATIC_TREE_DTD 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 topology 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 $1 \leqslant i \leqslant m-1$, 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 $1 \leqslant i \leqslant m-1$, 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:

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

In the above equation, we have used q instead of $q(t)$ for simplicity. Note that $0 < d_1 < d_2 < \cdots < d_{m-1}$ and $m \leqslant P$ for any PE on any topology, and that $0 \leqslant q(t) < 1$ 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:

$$\mathcal{M}_{\text{avg}} \leqslant \sum_{i=1}^{P-1} \left[(1-q) \cdot q^i \cdot i \right] = (1-q) \cdot q \cdot \sum_{i=0}^{P-1} \left[i \cdot q^{(i-1)} \right]$$

$$= (1-q)\cdot q\cdot \frac{d}{dq}\left[\sum_{i=0}^{P-1} q^i\right]$$

$$= (1-q)\cdot q\cdot \frac{d}{dq}\left[\frac{1-q^P}{1-q}\right]$$

$$= q\cdot \frac{1-q^{(P-1)}\,[P-(P-1)q]}{1-q}.$$

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 $\mathcal{M}_{\text{avg}} = O\left(\frac{1}{1-q}\right) = O(1)$, or the total average RESUME-message complexity is $O(M)$. Therefore, the average secondary-message complexity of STATIC_TREE_DTD under the assumptions of the theorem is $M_{\text{s,avg}} = \Theta(M+P)$. □

Note that a loaded PE is always busy except when after sending out some primary message it becomes idle and waits for an acknowledgment. In this case, the PE remains loaded and idle until it receives the pending acknowledgment 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 [16], 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 STATIC_TREE_DTD 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.

5.2.2. A practical technique for reducing message complexity

The constant associated with the average message complexity of our algorithm 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 non-zero) 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 $t_{\text{int}} = \Theta(B_{\text{opt}})$, 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 [4,17] (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 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 [12,13,19]. 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 12 and the above discussion that in most practical cases, the average message complexity of STATIC_TREE_DTD will be $\Theta(M+P)$ with a very small constant associated with it.

5.3. 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. 3). 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, $\Theta(MD+P)$ and $\Theta(M+P)$, respectively.

5.4. Overall performance summary

The following theorem summarizes our algorithm's performance:

Theorem 13. STATIC_TREE_DTD *has the following complexities for various performance metrics on an arbitrary target*

topology of diameter D and containing P PEs:

(1) *An optimal best-case detection delay of* $\Theta(1)$ *and finite optimal worst-case detection delays* T_d *and* T'_d *both equal in order terms to the time for an optimal broadcast in the target topology of* $\Theta(B_{opt})$*—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* $\Theta(MD+P)$ *and* $\Theta(M+P)$, *respectively*.
(3) *An optimal space complexity of* $\Theta(P)$.

Here M is the number of primary messages used by the primary computation.

6. Complexity analysis of some existing DTD algorithms and comparisons to STATIC_TREE_DTD

In this section, we first accurately characterize the performance of some related DTD algorithms and then compare them to STATIC_TREE_DTD.

6.1. Acknowledgment-based DTD algorithms

We now analyze the detection delays of three acknowledgment based algorithms. The algorithm of [4], like other acknowledgment-based algorithms, embeds a spanning tree in the target topology for DTD. 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 l, then PE j sends an acknowledgment to PE i when it becomes idle and has received acknowledgments for all primary messages (e.g., the one sent to l) 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 $2 \leqslant i \leqslant m-1$, then p_i will send an acknowledgment to p_{i-1} only after it receives one from p_{i+1} and becomes idle. A PE reports a STOP message 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 M acknowledgment messages and $P-1$ STOP messages and so its message complexity is $\Theta(M+P)$, which is worst-case optimal. However, since the M primary messages can form a sequence, as mentioned earlier, of $\Theta(M)$ PEs and the minimum possible worst-case secondary-message processing time for any acknowledgment-based algorithm is $\Theta(B_{opt})$, T_d is $\Theta(M+B_{opt})$, 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 is $\Theta(MD+B_{opt})$. The worst-case space complexity is $\Theta(M+P)$ to store labels of the PEs to which acknowledgments and STOPs need to be sent, and, similarly, the worst-case computational complexity is $\Theta(M+P)$ to process these secondary messages. From the above discussion and from Lemmas 2 and 3, we obtain:

Theorem 14. *The algorithm of* [4], *on an arbitrary target topology of diameter D and containing P PEs, has* $T_d = \Theta(M+B_{opt})$, $T'_d = \Theta(MD+B_{opt})$, *and* M_s, S_s, *and* C_s *all* $= \Theta(M+P)$, *where* $B_{opt} = \Omega(D)$ *and also* $B_{opt} = O(\min(Dd_{max}, P))$, d_{max} *is the maximum degree of a PE in the target topology, and M is the number of primary messages used by the primary computation. On k-ary n-cubes,* $T_d = \Theta(M+D)$ *and* $T'_d = \Theta(MD)$.

The algorithm of [17] is similar to that of [4] described above, except in the following respects. In [17], the set of loaded PEs at any time belongs 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 PE j and becomes a part of this tree with PE j as its parent, thus changing the structure of the tree. As in [4], PE i acknowledges receipt of message m from PE j only after it becomes free. However, unlike that in [4], all messages received by PE i when it is loaded are acknowledged as soon as it becomes idle, which helps improve its worst-case detection delay complexity while retaining the optimal worst-case message complexity feature of [4]. Such messages do not affect the structure of the tree of loaded PEs in [17]. Again, as in [4], 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. Clearly, $T_d = \Theta(B_{max})$ and $T'_d = \Theta(L'_{max,P-1} + B'_{max})$. The worst-case message and computational complexities are the same as those of [4]. The worst-case space complexity is $\Theta(P)$ to store the labels of as many PEs as to which STOPs need to be sent. From Lemmas 2 and 3 and the above analysis, we obtain:

Theorem 15. *The algorithm of* [17], *on an arbitrary target topology of diameter D and containing P PEs, has* $T_d = \Theta(B_{max})$, $T'_d = \Theta(L'_{max,P-1})$, $M_s = \Theta(M+P)$, $S_s = \Theta(P)$, *and* $C_s = \Theta(M+P)$, *where* $B_{max} = \Omega(D)$ *and* $B_{max} = O(P)$, $L'_{max,P-1} = \Omega(D^2+P)$ *and* $L'_{max,P-1} = O(PD)$, *and M is the number of primary messages used by the primary computation. On k-ary n-cubes,* $T_d = \Theta(P)$ *and* $T'_d = \Theta(PD)$.

One of the features of our algorithm is that its termination tree $\mathcal{T}_{opt}$ is static with $L_{opt} = \Theta(D)$ to optimize its detection delay. Based upon this idea, the algorithm of [28] works in a manner similar to that of [17] and "attempts" to improve its worst-case detection delay while retaining its optimal worst-case message complexity. The difference is that, unlike in [17], the termination tree structure, though still dynamic, is constrained in the following way. A termination subtree rooted at any child PE of

the main root PE is allowed to change as in [17] until it grows to a height beyond $2D$. At this point, the original subtree is split to reduce its height to less than D and a new subtree of height $\Theta(D)$ is created, the root PE of which is then attached directly as a child PE to the main root PE; this new subtree again has the same constraint on its height and may spawn new subtrees of its own when its height grows beyond $2D$. Note that although the root PEs of new subtrees created are child PEs of the main root PE, they may not necessarily be neighbors of the main root PE in the target topology, regardless of whether the primary computation uses near-neighbor or arbitrary-distance message passing. Vertices that are adjacent on a subtree, however, correspond to neighboring PEs in the target topology when the primary computation uses near-neighbor message passing. Therefore, the overall termination tree has a height of $\Theta(D)$ in the worst case. However, the worst-case detection delay does not necessarily improve compared to that of the algorithm of [17] because of two reasons. First, although subtrees created may initially have a height of $\Theta(D)$, they may shrink to even a single PE (i.e., to the root PE of the subtree) as PEs on it become free. Therefore, the number of subtrees that have their root PEs as the child PEs of the main root PE at the time of termination may be as high as $\Theta(P)$ in the worst-case, regardless of whether near-neighbor or arbitrary-distance message passing is used by the primary computation. This means that the worst-case secondary-message processing time at the main root PE is $\Theta(P)$. Second, in the arbitrary-distance message-passing case, adjacent vertices on the termination tree may correspond to PEs that are $\Theta(D)$ distance apart (because $L'_{\max,D} = \Theta(D^2)$ from Lemma 2). The secondary message, space, and computational complexities of the algorithm are the same as that of [17]. Therefore, from the above observations and from Lemmas 2 and 3, we obtain:

Theorem 16. *The algorithm of* [28], *on an arbitrary target topology of diameter D and containing P PEs, has* $T_d = \Theta(P)$, $T'_d = \Theta(P + D^2)$, $M_s = \Theta(M + P)$, $S_s = \Theta(P)$, *and* $C_s = \Theta(M + P)$, *where M is the number of primary messages used by the primary computation. On k-ary n-cubes,* $T'_d = \Theta(P^2)$ *for* $n = 1$ *and* $T'_d = \Theta(P)$ *for* $n \geqslant 2$.

Therefore, T_d for the algorithm of [28] is at least as high as that of [17], but may be worse, and T'_d may be as high as that of [17], but may be better.

6.2. Message-counting based DTD algorithm

Finally, we consider the algorithm in [14] which uses a message-counting technique. It uses a marker that traverses a cycle with C PEs on it such that the cycle spans all PEs in the target topology—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 target system. 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, at the start of a visit to a PE i, $recp_i = 0$, and $sntm = recm$ at the end of the last visit in the sequence, it signals termination. This algorithm can have counter-overflow problems. The worst-case 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, the complexity of which is given in the next lemma.

Lemma 17. *A cycle containing all vertices can be found in any connected graph with P vertices such that the number of edges on the cycle is* $\Theta(P)$.

Proof. Such a cycle can be formed, for instance, as follows. Find a tree by a breadth-first or depth-first search in the given graph. Then the traversal path (which includes backtracks) corresponding to a depth-first search in this tree from some root vertex is a cycle since the search starts at the root vertex, visits all other vertices, and then finishes at the root vertex. Moreover, since each edge in the tree is traversed exactly twice, and the tree has $P - 1$ edges, the cycle has $2(P - 1)$ edges. □

Therefore, the worst-case detection delay is $\Theta(P)$. Also, in the worst-case, the marker will have to go around the cycle once per primary-message transmitted, each of which has some computation associated with it. Therefore, we obtain:

Theorem 18. *The algorithm of* [14], *on an arbitrary target topology containing P PEs, has* $T_d = \Theta(P)$, $T'_d = \Theta(P)$, $M_s = \Theta(MP)$, $S_s = \Theta(P)$, *and* $C_s = \Theta(MP)$, *where M is the number of primary messages used by the primary computation.*

6.3. Comparisons between DTD algorithms

In Table 1, we compare the performance of our algorithm with the optimal case and with four of the most efficient algorithms proposed to date [4,17,28,14] that were analyzed in the previous two subsections. 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 factor of D more than optimal, which is obviously because of the higher message complexity. However, as proved in Theorem 12 and from the discussion following the theorem, the average message complexity, and hence the average computational complexity, for most applications will be $\Theta(M + P)$ with a small constant factor. Furthermore, the constant factor in this complexity can be greatly reduced further by using counters as discussed in Section 5.2.2. 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:

Table 1
Performance comparison of different DTD algorithms on a target system with arbitrary topology in terms of: (1) worst-case detection delay T_d when message passing is between only neighboring PEs; (2) worst-case detection delay T'_d when message passing is between arbitrary PEs; (3) worst-case message complexity M_s; (4) average message complexity $M_{s,avg}$, (5) worst-case space complexity S_s; (6) worst-case computational complexity C_s; and (7) average computational complexity $C_{s,avg}$

Algorithms	Optimal case	Chandrasekaran et al. [4]	Lai et al. [17]	Mittal et al. [28]	Kumar [14]	Our algorithm
T_d	$\Theta(B_{opt})$	$\Theta(M + B_{opt})$	$\Theta(B_{max})$	$\Theta(P)$	$\Theta(P)$	$\Theta(B_{opt})$
T'_d	$\Theta(B_{opt})$	$\Theta(MD + B_{opt})$	$\Theta(L'_{max,P-1})$	$\Theta(P + D^2)$	$\Theta(P)$	$\Theta(B_{opt})$
M_s	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(MP)$	$\Theta(MD + P)$
$M_{s,avg}$	$O(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$O(MP)$	$\Theta(M + P)$
S_s	$\Theta(P)$	$\Theta(M + P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$
C_s	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(MP)$	$\Theta(MD + P)$
$C_{s,avg}$	$O(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$\Theta(M + P)$	$O(MP)$	$\Theta(M + P)$
Relevant bounds	$B_{opt} = \Omega(D)$, $B_{opt} = O(\min(Dd_{max}, P))$		$B_{max} = \Omega(D)$, $B_{max} = O(P)$		$L'_{max,P-1} = \Omega(D^2 + P)$, $L'_{max,P-1} = O(PD)$	

D denotes the diameter, d_{max} the maximum degree of a PE, and P the number of PEs in the target system, B_{opt}, B_{max}, and $L'_{max,P-1}$ are properties of the target system topology defined in Section 2.2, and M denotes the number of primary messages used by the primary computation.

Table 2
Performance comparison of different DTD algorithms on some popular target topologies in terms of: (1) worst-case detection delay T_d when message passing is between only neighboring PEs and (2) worst-case detection delay T'_d when message passing is between arbitrary PEs

Algorithms		Optimal case	Chandrasekaran et al. [4]	Lai et al. [17]	Mittal et al. [28]	Kumar [14]	Our algorithm
k-ary n-cube	T_d	$\Theta(nk)$	$\Theta(M + nk)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(nk)$
Torus or mesh	T'_d	$\Theta(nk)$	$\Theta(Mnk)$	$\Theta(Pnk)$	$\Theta(P + n^2k^2)$	$\Theta(P)$	$\Theta(nk)$
Linear array	T_d	$\Theta(P)$	$\Theta(M + P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$
or ring	T'_d	$\Theta(P)$	$\Theta(MP)$	$\Theta(P^2)$	$\Theta(P^2)$	$\Theta(P)$	$\Theta(P)$
2-D Mesh	T_d	$\Theta(P^{1/2})$	$\Theta(M + P^{1/2})$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P^{1/2})$
or Torus	T'_d	$\Theta(P^{1/2})$	$\Theta(MP^{1/2})$	$\Theta(P^{3/2})$	$\Theta(P)$	$\Theta(P)$	$\Theta(P^{1/2})$
3-D Mesh	T_d	$\Theta(P^{1/3})$	$\Theta(M + P^{1/3})$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P^{1/3})$
or Torus	T'_d	$\Theta(P^{1/3})$	$\Theta(MP^{1/3})$	$\Theta(P^{4/3})$	$\Theta(P)$	$\Theta(P)$	$\Theta(P^{1/3})$
Hypercube	T_d	$\Theta(\log P)$	$\Theta(M + \log P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(\log P)$
	T'_d	$\Theta(\log P)$	$\Theta(M \log P)$	$\Theta(P \log P)$	$\Theta(P)$	$\Theta(P)$	$\Theta(\log P)$

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 [14], increases greatly when message passing between arbitrary PEs is allowed, while it remains the same for our algorithm (see Section 5.1). (2) M_s, C_s: we have shown that the message and computational complexities of our algorithm for most applications are $\Theta(M + P)$ on the average, same in order terms as other message-efficient algorithms [4,17,28], but with a very small constant associated with it. The message complexity of [14] (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 [4] and equal to those of [17,28,14]. Thus, overall, our algorithm has almost optimal performance that is much better than those of other algorithms.

7. Conclusion

In this paper, we presented a common framework that can be used to accurately analyze the detection delay of DTD algorithms, especially those based on acknowledgments. We also presented our new STATIC_TREE_DTD algorithm that, unlike other acknowledgment-based algorithms, employs a static termination tree structured so as to minimize the sum of secondary-message communication and processing times on the target topology. We analyzed our and related DTD algorithms accurately using a number of relevant metrics for an arbitrary topology and for k-ary n-cubes in particular. We found that while previous DTD algorithms may be optimal with respect to one performance metric, they are inefficient with

regard to other metrics. In particular, all previous methods have a worst-case detection delay, the most important metric, of $\Omega(P)$ on most topologies, where P is the total number of PEs. 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. STATIC_TREE_DTD, on the other hand, is optimal or near-optimal with respect to all relevant performance metrics: its detection delay is optimal ($O(D)$ on k-ary n-cubes), message and computational complexities are $O(MD + P)$ ($\Theta(M + P)$ on the average for most applications—the same as other message-efficient algorithms [4,17,28]), and space complexity is $\Theta(P)$ (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 [4,17]. Furthermore, unlike some previous DTD algorithms [4,5,26], STATIC_TREE_DTD does not require links to support FIFO message communication for correct operation.

References

[1] G.S. Almasi, A. Gottlieb, Highly parallel computing, Benjamin, Cummings, Redwood City, CA, 1994.

[2] R. Atreya, N. Mittal, V.K. Garg, Detecting locally stable predicates without modifying application messages, in: Proceedings of the 7th International Conference on Principles of Distributed Systems, December 2003, pp. 20–33.

[3] A.H. Baker, S. Crivelli, E.R. Jessup, An efficient parallel termination detection algorithm, Internat. J. Parallel Emergent Distributed Systems 21 (4) (2006) 293–301.

[4] S. Chandrasekaran, S. Venkatesan, A message-optimal algorithm for distributed termination detection, J. Parallel Distributed Comput. 8 (1990) 245–252.

[5] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Systems 3 (1) (1985) 63–75.

[6] K.M. Chandy, J. Misra, Asynchronous distributed simulation via a sequence of parallel computations, Comm. ACM 24 (4) (1981) 198–205.

[7] K.M. Chandy, J. Misra, How processes learn, Distributed Comput. 1 (1) (1986) 40–52.

[8] W.J. Dally, Performance analysis of k-ary n-cube interconnection networks, IEEE Trans. Comput. 39 (6) (1990).

[9] P.K. Dash, R.C. Hansdah, A fault-tolerant distributed algorithm for termination detection using roughly synchronized clocks, in: Proceedings of the 1997 International Conference on Parallel and Distributed Systems, 1997, pp. 736–743.

[10] N. Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[11] M. Demirbas, A. Arora, An optimal termination detection algorithm for rings, Technical Report OSU-CISRC-2/00-TR05, The Ohio State University, February 2000.

[12] S. Dutt, N.R. Mahapatra, Parallel A* algorithms and their performance on hypercube multiprocessors, in: Seventh International Parallel Processing Symposium, 1993, pp. 797–803.

[13] S. Dutt, N.R. Mahapatra, Scalable load balancing strategies for parallel A* algorithms, J. Parallel Distributed Comput. 22 (3) (1994) 488–505.

[14] D. Kumar, Development of a class of distributed termination detection algorithms, IEEE Trans. Knowledge Data Eng. 4 (2) (1992).

[15] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to parallel computing: design and analysis of algorithms, Benjamin, Cummings Publishing Company, Redwood City, CA, 1994.

[16] V. Kumar, A. Gupta, Analyzing scalability of parallel algorithms and architectures, J. Parallel Distributed Comput. 22 (3) (1994) 379–391.

[17] T.-H Lai, Y.-C. Tseng, X. Dong, A more efficient message-optimal algorithm for distributed termination detection, in: Proceedings of the Sixth International Parallel Processing Symposium, March 1992, pp. 646–649.

[18] T.-H. Lai, L.-F. Wu, An $(N - 1)$-resilient algorithm for distributed termination detection, in: Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, 1992, pp. 274–281.

[19] N.R. Mahapatra, S. Dutt, New anticipatory load balancing strategies for parallel A* algorithms, in: P.M. Pardalos, M.G. Resende, K.G. Ramakrishnan (Eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science—Parallel Processing of Discrete Optimization Problems, vol. 22, 1995, pp. 197–232.

[20] N.R. Mahapatra, S. Dutt, Sequential and parallel branch-and-bound search under limited-memory constraints, in: P. Pardalos (Ed.), The IMA Volumes in Mathematics and its Applications—Parallel Processing of Discrete Problems, vol. 106, Springer, New York, 1999, pp. 139–158.

[21] N.R. Mahapatra, S. Dutt, An efficient delay-optimal distributed termination detection algorithm, Technical Report# 2001-16, Department of Computer Science & Engineering, University at Buffalo, The State University of New York, Buffalo, NY, 2001. Available at: ⟨http://www.cse.buffalo.edu/tech-reports/2001-16.ps⟩.

[22] J. Matocha, Distributed termination detection in a mobile wireless network, in: Proceedings of the 36th Annual Southeast Regional Confernece, 1998, pp. 207–213.

[23] J. Matocha, T. Camp, A taxonomy of distributed termination detection algorithms, J. Systems Software 43 (3) (1998) 207–221.

[24] F. Mattern, Efficient algorithms for distributed snapshots and global virtual time approximation, J. Parallel Distributed Comput. 18 (4) (1993) 423–434.

[25] J. Mayo, P. Kearns, Distributed termination detection with roughly synchronized clocks, Inform. Process. Lett. 52 (1994) 105–108.

[26] J. Misra, Detecting termination of distributed computations using markers, in: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, August 1983, pp. 290–294.

[27] N. Mittal, F.C. Freiling, S. Venkatesan, L.D. Penso, Efficient reduction for wait-free termination detection in a crash-prone distributed system, in: Proceedings of the 19th International Conference on Distributed Computing, October 2005, pp. 93–107.

[28] N. Mittal, S. Venkatesan, S. Peri, Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies, in: Proceedings of the 18th Symposium on Distributed Computing, October 2004, pp. 290–304.

[29] S. Peri, N. Mittal, On termination detection in an asynchronous distributed system, in: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, September 2004, pp. 209–215.

[30] S. Peri, N. Mittal, Improving the efficacy of a termination detection algorithm, J. Inform. Sci. Eng., accepted for publication. Available at: ⟨https://proxy.utdallas.edu/~sxp026300/index_files/Jour-JISE-Transtd.pdf⟩.

[31] S.P. Rana, A distributed solution of the distributed termination problem, Inform. Process. Lett. 17 (1) (1983) 43–46.

[32] G.-C. Roman, J. Payton, A termination detection protocol for use in mobile ad hoc networks, Automated Software Eng. J. 12 (1) (2005) 81–99.

[33] J.M. Rosario, A.N. Choudhary, High-performance I/O for massively parallel computers: problems and prospects, Computer (1994) 59–68.

[34] Y. Saad, M.H. Schultz, Data communication in hypercubes, J. Par. Distr. Comput. 6 (1989) 115–135.

[35] P.A. Steenkiste, A systematic approach to host interface design for high-speed networks, Computer (1994) 47–58.

[36] G. Tel, F. Mattern, The derivation of distributed termination detection algorithms from garbage collection schemes, ACM Trans. Programming Languages Systems 15 (1) (1993) 1–35.

[37] Y.-C. Tseng, Detecting termination by weight-throwing in a faulty distributed system, J. Parallel Distributed Comput. 25 (1995) 7–15.
[38] S. Venkatesan, Reliable protocols for distributed termination detection, IEEE Trans. Reliability 38 (1) (1989) 103–110.
[39] X. Wang, J. Mayo, A general model for detecting distributed termination in dynamic systems, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.

Nihar R. Mahapatra received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Delhi, India, in 1990 and the M.S. and then the Ph.D. degree from the University of Minnesota, Twin Cities, in 1996. He is currently an Associate Professor of Electrical and Computer Engineering at Michigan State University. His research interests include computer architecture and VLSI and parallel and high-performance computing. In these areas, he has published more than 60 papersin leading refereed journals, conferences, and workshops, and has also contributed invited papers. A paper he co-authored was nominated for the best paper award at HPCA 2005. He has given invited research talks in academia, industry, conferences, and workshops. His Ph.D. dissertation was nominated for the ACM doctoral dissertation award from the University of Minnesota. His research has been funded by NSF, AFRL, AFOSR, IBM, Lockheed Martin Corp., and intramural grants. He has served on several NSF proposal review panels, as a processor architecture track co-chair in ICCD 2003, as a session chair in ICCD 2003–2004 and SOCC 2002 and 2004, as a program committee member in GLSVLSI 2003–2004, HiPC 1999, ICCD 1997 and 2003–2004, IPDPS 2001, ISVLSI 2004–2007, SOCC 2002–2007, and several others, and as a referee for numerous journals and conferences.

Shantanu Dutt received a B.E. degree in Electronics and Communication Engineering from the M.S. University of Baroda, Baroda, India in 1983, an M.Tech. degree in Computer Engineering from the Indian Institute of Technology, Kharagpur, and a Ph.D. degree in Computer Science and Engineering from the University of Michigan, Ann Arbor, in 1990. He is currently an associate professor at the Department of Electrical and Computer Engineering, University of Illinois, Chicago. In the past he has had two short stints in industry, at CMC Research at Secunderabad, India in 1985 as an R&D engineer, working on relational database design (the topic of his M.Tech. thesis), and as a Senior Consultant in the area of CAD, at Cadence Design Systems, San Jose, CA, during the summer of 1996.

Prof. Dutt was awarded a Research Initiation Award by the National Science Foundation. His current technical interests include CAD for sub-micron VLSI circuits, fault-tolerant computing and testing for emerging VLSI technologies. His research has been funded by NSF, DARPA, AFOSR and companies like Xilinx and Intel. He has published around 70 papers in well-recognized archival journals and refereed conferences in all the above areas. He has received a Best-Paper award at the “Design Automation Conference (DAC)”, 1996, a Most-Influential-Paper award from the “Fault-Tolerant Computing Symposium (FTCS)” in 1995 for a paper published in FTCS 1988 and a Best-Paper nomination at the ”Design Automation Conference”, 2004. He has also been one of two Featured Speakers at the “International Conference on Computer-Aided Design (ICCAD)” 2006.

He has contributed an invited articles in the ”Wiley Encyclopedia of Electrical and Electronics Engineering” and the ”Electrical Engineering Handbook” from Academic Press, and an invited paper at DIMACS. He has been on several NSF review panels and the technical program committees of conferences on fault-tolerant computing, parallel processing and VLSI CAD. Recently he was a co-chair of the Tools and Methodology track of ICCD’04 and on the technical program committees of ICCD 2003 and GLSVLSI 2004–2007.