Revisiting Credit Distribution Algorithms for Distributed Termination Detection

George Bosilca*, Aurélien Bouteiller*, Thomas Herault*, Valentin Le Fèvre†, Yves Robert*‡, Jack Dongarra*
*University of Tennessee Knoxville, TN, USA
†Barcelona Supercomputing Center, Spain
‡Laboratoire LIP, ENS Lyon, France

***Abstract*—This paper revisits distributed termination detection algorithms in the context of High-Performance Computing (HPC) applications. We introduce an efficient variant of the Credit Distribution Algorithm (CDA) and compare it to the original algorithm (HCDA) as well as to its two primary competitors: the Four Counters algorithm (4C) and the Efficient Delay-Optimal Distributed algorithm (EDOD). We analyze the behavior of each algorithm for some simplified task-based kernels and show the superiority of CDA in terms of the number of control messages.**

***Index Terms*—Termination detection, credit distribution algorithms, task-based HPC application, control messages.**

I. Introduction

A distributed application is *terminated* if all processes have completed the computations assigned to them and no message is in transit within the interconnection network. Termination detection is a fundamental issue for distributed systems, because – for dynamic applications – no process has complete knowledge of the global configuration (the state of all processes and of the network) [1]. In particular, an idle process may be reactivated by a message from another process, complete its new assignment, send some work orders to be completed by remote processes, and then become idle again and so on. Many *active-to-idle* and *idle-to-active* transitions can take place before the application eventually terminates. Since the pioneering work of Dijkstra, Scholten, and Francez [2], [3], countless algorithms have been proposed for termination detection.

Many high-performance computing (HPC) applications can rely on straightforward techniques for termination detection. For instance, many dense or sparse factorization algorithms terminate when the bottom-right diagonal element of the matrix has been updated, and termination can safely be declared right after the completion of that last operation. More generally, many HPC applications are structured as a task graph with all dependencies statically known before execution. Termination can safely be declared once all exit tasks (tasks without any successor task) of the graph have been completed. However, there are also many HPC applications whose task graphs are dynamically updated during the execution: the application task graph is data dependent, and new tasks may be created depending on the value of the output of another task. Typical examples are partial differential equation (PDE) schemes, where the necessary degree of refinement is dictated by the physics of the simulated material. For all of these applications, a distributed termination detection algorithm must be implemented.

Our main contribution is the design of a new termination detection algorithm that is specialized for HPC platforms. We adopt a simplified but realistic model for such platforms. For instance, message loss and re-ordering are routinely managed by the network layer (e.g. MPI, OpenSHMEM,etc.), hence we can safely design algorithms that benefit from these features and assume that messages are delivered in FIFO order. We focus on performance at scale and aim at minimizing the overhead incurred by the termination detection algorithm on the application. Clearly, detection algorithms that use many control messages will delay, or add extensive management of, application messages and will be detrimental to application progress. In this work, we consider different classes of termination detection algorithms and evaluate their overhead in terms of the number of control messages that are generated.

We distinguish and compare three main classes of algorithms for termination detection. First, many algorithms use ascending and descending waves of control messages, and we discuss the Four Counter algorithm (4C) – a state-of-the-art wave algorithm – in Section III. The Credit Distribution Algorithms (CDA) are another set of algorithms proposed independently by Huang [4] and Mattern [5]. These algorithms are also known as *weight-throwing algorithms*, and they use a controlling agent that initially distributes some credit to all processes. When sending an application message, a process keeps a fraction of its current credit and transfers the remaining fraction through the message; upon reception of a message, the credit carried by the message is added to the credit of the receiving process. Finally, when becoming idle, a process returns its credit to the controlling agent. The controlling agent declares termination when all of the initially distributed credit has been returned to it. We introduce the original algorithm, "Huang's CDA" (HCDA), discuss several existing variants, and propose a novel CDA algorithm dedicated to HPC platforms in Section III. Finally, a more recent class of algorithms, Efficient Delay-Optimal Distributed (EDOD) termination detection algorithms [6], requires that a control message acknowledging primary messages reception is sent by the receiver of each application message back to the sender; this is to ensure that the sender can be safely declared terminated once all of its messages have been acknowledged. These control messages go up and down a control binary

tree – independent of the application communications. EDOD is carefully designed to minimize the latency of termination detection, and we describe it in more detail in Section III.

Our main contribution is the design and implementation of a novel CDA variant that drastically improves performance, under the constraints of an HPC system, with a more conservative but mathematically accurate credit management system, where the borrowing operation can be satisfied by a neighbor process with more abundant resources. We evaluate the algorithms in terms of the number of control messages, through a theoretical analysis for the token ring application, and through simulations for synchronous tree-based task systems. As stated above, we focus on the number of messages generated by each algorithm as the key indicator of performance and overhead.

The paper is organized as follows. In Section II, we present motivating applications and systems that require termination detection. We review 4C, HCDA, and EDOD in Section III. We introduce our new CDA algorithm in Section IV and provide a theoretical comparison with HCDA, 4C and EDOD in Section V. We survey related work in Section VI. We provide concluding remarks and directions of future work in Section VII.

II. Dynamic Applications and Termination Detection

Termination detection is often implicit or trivial in regular, static applications, for which the control-flow of the application and/or the initial load balance of the work is sufficient to decide, locally, the termination. The issue becomes more crucial for dynamic applications expressed over asynchronous programming paradigms, for which the total amount of work is data dependent—and therefore remains unknown until completion. Here, we focus on efficiently detecting that an application producing supplementary work and messages, from process-local criterion, is globally complete.

To illustrate the concept, consider the example of k-dimensional trees that represent approximations of multidimensional functions and operators. Consider for $k = 1$ a function, $f(x)$, that should be approximated over a domain, $[A, B]$. A 1-dimensional tree is used to approximate the values of f by splitting $[A, B]$ into subdomains, $[a_i, b_i)$. For each subdomain, a leaf in the tree is created that carries a single value: the average of f in that subdomain, $\int_a^b f(x)dx/(b-a)$. The size of the subdomain (and thus the quality of the approximation) is set by selecting the depth of the leaves in the tree. Figure 1 illustrates this approach.

A task-based approach to create such representations is used in the Multiresolution ADaptive Numerical Environment for Scientific Simulation (MADNESS) [7], which is a high-level software environment for the resolution of integral and differential equations in multiple dimensions using adaptive and fast harmonic analysis methods with guaranteed precision. The operation of creating a tree that represents a given function in a given domain for a target precision is called a "projection." A natural and efficient algorithm to implement the projection consists of walking down the tree in parallel, with each task instantiating a node and deciding locally if a given node in the tree is refined enough to reach the target precision, in which case it is defined as a leaf. If not, its $2k$ children are spawned to increase the refinement. As the algorithm proceeds with refining the nodes, a mapping defines which tasks/nodes are held by which process of the parallel application. Depending on the targeted function, refinement, and data distribution, a process may be done with all current tasks but still receive more tasks to instantiate higher refinements at any time—until all processes are finished with all tasks.

A naive approach to detect termination for this algorithm would be to wait for the entire subtree to complete before letting the task complete, every time a task spawns refinement nodes. This approach has multiple obvious drawbacks: if the wait monopolizes computing resources, a starvation will occur when the number of nodes in the k-dimensional tree exceeds the number of computing elements. Even if better strategies are implemented to avoid this resource consumption, control information about the completion of each task must be sent to the process holding the parent node, thereby introducing large delays and costs. Because a process may receive work at any time, local observations that the number of tasks to complete has reached zero is not sufficient to decide termination, and a distributed termination algorithm is necessary.

This issue occurs in many tree-based algorithms and is a key part of composition. For example, occurring frequently in MADNESS algorithms, multiple functions must be projected in order to be derived, summed, multiplied and integrated to compute a solution to the final problem. To reduce overheads, all of these operations should start with maximum concurrency, but knowledge about the completion of dependent operations is necessary to ensure the correctness of the result. Distributed termination detection algorithms rely on observing the activity of the processes, as well as the injection and delivery of application messages, sometimes modifying them to piggyback information. Since these roles are assigned to the runtime system, it is also natural to assign the role of detecting the termination of global operations to the runtime environment.

III. Algorithms for Termination Detection

Sections III-B to III-D detail the main features of the three primary detection termination algorithms from the literature: 4C (waves with in-transit message detection), EDOD (acknowledged primary messages), and HCDA (Huang's credit distribution), which we contrast with our own CDA algorithm in Section IV. Beforehand, in Section III-A, we review the system model common to all algorithms.

A. System Model

We consider a distributed system comprised of a set of $\mathcal{P}$ processes with an independent clock and a local memory. The processes are connected through an asynchronous interconnection network capable of carrying messages in 1-port duplex mode with an arbitrary, but finite, delay. Processes and messages are considered here in the general sense: processes

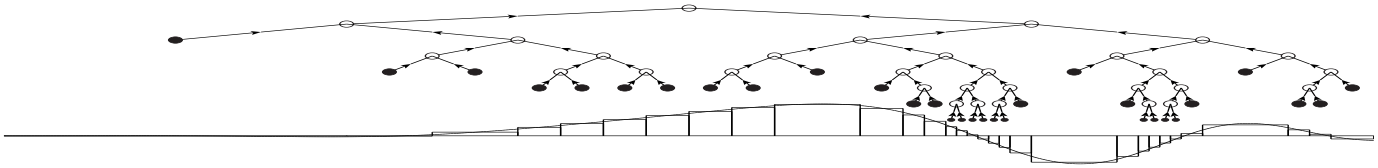

Figure 1: Sample application: $1-$dimensional tree whose refinement locally depends on the slope of the target function.

may employ internal shared-memory parallelism (which is abstracted from the model), and remote memory accesses can be considered as asynchronous messages. The processes and network are reliable, and we assume that the interconnection network is complete; that is to say: any process may send a direct message to any other processes. We also assume that messages may not overtake; in other words, the network is assumed to be FIFO, or the network library manages ordering and reemission as necessary (e.g. MPI). Although not required for correcting the algorithms, these assumptions simplify performance analysis.

A parallel workload executes internal actions on the processes, either executing a task or creating a new task. Task mapping (to processes) is determined by an application-provided mapping function, and successor tasks may be mapped onto a remote process, which entails the emission of a message. When the destination of a message is the local process, it is considered a local action.

In such a distributed system, we consider the termination detection problem. Termination detection is achieved when all processes know that every process has completed the workload. More formally, a process is still considered *active* when it has pending actions, including when it is executing a task, has scheduled tasks to execute locally, or has pending emissions to perform. When a process does not have any further pending local actions, it becomes *idle*. A process may exit from the *idle* state and return to the *active* state only when it receives a message (i.e., tasks can be only created upon completion of another task). Without loss of generality, we consider that, initially, one process contains a startup task (there is a trivial transformation to render any workload with multiple initial tasks compliant). The termination of the workload is a global stable state that is reached when, in a global snapshot [8], every process is in the *idle* local state, and there are no in-transit messages (since, otherwise, these in-transit messages could create work for some of the idle processes). The termination is detected when every process has been informed that this global state has been reached.

Termination detection algorithms are thus distributed algorithms that *observe* that the global state has been reached and then *announce* it to all processes. In some algorithms, the detection and announce phases may be merged or overlapped. The distributed termination detection algorithm likely requires the exchange of *secondary messages* (i.e., supplementary control messages added to the *primary messages* generated by the parallel workload). These secondary messages allow the process states to be gathered/reported to a centralized entity or be part of a termination broadcast.

B. *The* 4C *Wave Algorithm*

In wave algorithms, when a process becomes idle, it initiates a wave to verify the state of other processes in the system. The wave crosses the network and collects the status of individual processes and their communication channels at some process – either at the initiator or at some external entity. That process then inspects the collected global state to ascertain when the global termination state has been reached. For example, a process that switches from active to idle may initiate a distributed snapshot. The snapshot permits to detect in-transit messages, i.e., messages that have been emitted before the beginning of the wave, but received after its beginning at another process. Thus, after completing the snapshot, a process can report to the announcer if it was active, or if it detected an in-transit message at the logical time of the snapshot. Unfortunately, this approach requires performing a large number of waves. Specifically, one wave for every process's transition from active to idle, which – in the worst case – may result in as many waves as primary messages. The approach also suffers from a large termination detection delay.

The 4C wave algorithm, which has seen some practical uses in [7], can avoid some of these caveats. In this algorithm, processes are organized along a secondary tree overlay, and the root of that tree announces when termination is detected. Every process, p, counts how many primary messages it has sent, s_p, and received, r_p. It also maintains two accumulating counters, σs_p^i and σr_p^i, initially set to 0, representing the cumulative number of primary messages sent and received by all processes in the subtree rooted at p – as collected during wave i.

Independent of their idle or active state, processes can be in the *UP* or *DOWN* state (UP initially). When a leaf in the tree becomes idle in the UP state, it enters the DOWN state and sends its two counters to its parent in a STOP message. When a node in the tree receives a STOP message from its children, it accumulates the counters. When it becomes idle in the UP state and has received a STOP message from all of its children, it enters the DOWN state and propagates the counters to its parent.

When the root enters the DOWN state, it compares σs_{root}^i, σr_{root}^i, σr_{root}^{i-1}, and σs_{root}^{i-1}. If they are all equal, it broadcasts the termination; otherwise, it sends down a *REPEAT* message (propagated by all) that initiates the nodes' transition from the DOWN state to the UP state (thus starting another wave).

Comparing σs_{root}^i and σr_{root}^i is not a sufficient condition for termination, as one has to account for orphan messages, i.e., messages emitted by some process *after* the wave and received by some other process *before* the wave. If the wave is crossed by orphan messages, the reception is counted in the

accumulator, σr^i_{root}, but its emission is not. Thus, an orphan message may cancel the difference, $\sigma r^i_{root} - \sigma s^i_{root}$, even when an in-transit message is present, which would render the algorithm incorrect. If the value of σs^i_{root} remains constant during two consecutive waves, then the prior wave had no orphan messages, hence the counter comparison is a valid estimator for the absence of in-transit messages.

C. Optimal Delay Algorithm

Mahapatra and Dutt [6] note that many termination detection algorithms focus on optimizing for the minimal number of secondary messages but often exhibit poor detection delay on commonly used primary communication patterns, like *k-ary n-cubes*, especially when considering a bounded port model, where message management time is considered. For this reason, the authors focus on designing an algorithm the purpose of which is to attain the optimal detection delay on arbitrary primary communication patterns.

Their EDOD algorithm requires that primary messages be acknowledged by secondary messages to prevent premature termination announcements. Their algorithm also uses a secondary static spanning tree to reduce status change messages to the root and to broadcast the termination announcement. The secondary overlay can be (but does not have to be) extracted as a subset of the primary communication topology when it is known in advance. The root is then selected as a central process at a minimal distance to all leaf processes.

When the root process becomes idle, it announces the termination. When a non-root process becomes idle, it sends a STOP message to its parent. A process cannot become idle until it receives a STOP message from all of its children.

During the normal course of the computation, the algorithm counts the outgoing primary messages. A process cannot become idle until it receives a secondary *acknowledge* message for every outgoing primary message. When receiving a primary message, the receiver, r, may be active or idle. When r is active, it acknowledges the reception using a direct $\text{ACK}_{s,r}$ secondary message to the sender s. When r is idle, it becomes active and sends a $\text{RESUME}_{s,r}$ message to its parent. The parent may receive the $\text{RESUME}_{s,r}$ message when it is active or idle. When an idle parent receives a $\text{RESUME}_{s,r}$ message from a child, it becomes active, forgets the reception of the STOP message from that child, and forwards the $\text{RESUME}_{s,r}$ message to its parent. When an active parent receives a $\text{RESUME}_{s,r}$ message from a child, it forgets the reception of the previous STOP message from that child, and sends the $\text{ACK}_{s,r}$ to r, following the inverse path from the $\text{RESUME}_{s,r}$ message, then r sends $\text{ACK}_{s,r}$ to s directly. In effect, delaying the $\text{ACK}_{s,r}$ message prevents the root of the subtree containing s and r from becoming idle when a potential $\text{RESUME}_{s,r}$ message is canceling the STOP-message-induced actions on r's ancestors.

D. Credit Distribution Algorithms

In a credit distribution algorithm (e.g., HCDA), as originally proposed independently by Huang [4] and Mattern [5], an initiator controlling agent starts the computation with C_{total} total credit, and the initiator distributes the credit among processes according to the initial activity of the processes. During execution, messages carry credit between processes: when a process sends a message, it sends a fraction of its credit along with the message and keeps a fraction of the credit for itself. When a process receives a message, it adds the message-carried credit to its own credit stash. When a process becomes idle, it returns its entire stash of credit to the initiator. From there, the initiator process can detect the termination of all other processes when it again has C_{total} credits. Note that, as usual, an idle process may reset to active as a result of receiving a message. In this case, the process transitioning from an idle to an active state inherits the credit that has been carried in the in-transit message, thus guaranteeing that the initiator misses a fraction of the C_{total} credits for as long as any in-transit or active processes remain.

This approach is elegant in theory, but it suffers from multiple drawbacks that hinder its implementation. In non-infinite precision arithmetic, the HCDA algorithm is subject to an underflow problem when dividing the weight into two halves upon message emission. To partially alleviate this problem, Mattern [5] suggests using only credits of the form $X = 2^{-Y}$, where Y is an integer, and to encode $Y = -\log_2 X$ to represent X. This requires some modifications to the algorithm, outlined below.

- Use 2^{-q} as the initial local credit, where $2^{q-1} < \mathcal{P} \leq 2^q$, and total credit is now $C_{total} = \mathcal{P}2^{-q}$.
- An active node receiving a basic message returns the message-carried credit to the collecting agent, instead of storing it locally, to keep its own summing simple.

Then, all message weights have a weight, 2^{-Y}, for some Y, and sending a message splits the weight by incrementing Y. However, the complete summation is delegated to the controlling agent rather than eliminated, and many secondary control messages are needed to return the non-locally summed credit to the controlling agent.

Another variant suggested in [1, Ch. 6] allows a node without any remaining credit to create its own credit currency and start a weight-throwing termination detection subcall. Then, that node returns its weight to the initiator when it has become passive and its subcall has terminated. The weights originating from the initiator and from the node must be maintained separately. Again, this variant incurs additional control overhead and extra delays. In Section IV, we discuss how we build upon the basic HCDA strategy to design an algorithm suitable for extreme-scale, distributed HPC systems in a manner that avoids producing a large number of secondary credit return messages and operates without messaging delays.

IV. CDA for HPC

We expand on the classical CDA algorithm with original considerations for HPC platforms executing large-scale, distributed dataflow programs. The major challenge with CDA is the credit attrition resulting from the non-infinite divisibility

of the credit representation. Our CDA algorithm strives to achieve a low number of control messages while reducing the disruption of the exchange of primary messages. In our CDA algorithm, credit is represented as integer values (i.e., credit is not infinitely divisible but can be summed efficiently without arbitrary precision arithmetic). During the initial state, credit is distributed equally among nodes. Each process starts with an initial credit of value, C_{init}, known by all. The total amount of credit distributed initially is thus $C_{total} = \mathcal{P}C_{init}$. Note that, in certain applications, not all processes are initially active, and an application-specific policy may have achieved a more optimal initial distribution (e.g., by dividing the credit among initially active processes) but at the expense of losing generality. Initial credit is computationally generated and requires no secondary messages to be distributed.

When a process becomes idle, it returns its credit to the controlling agent with a *FLUSH* secondary message. This strategy has two drawbacks: (1) it increases the number of control messages, significantly in the worst case; and (2) it accelerates the rate of global attrition of credit in non-initiator processes by removing the flushed credit from circulation (hence increasing the chance that some active process will run out of credit). In primary algorithms executed as a dataflow, the locally visible horizon of tasks scheduled in the runtime can be leveraged to detect that an outgoing message is terminal, that is, the last message sent before a transition to idle.

We observe that sending the whole locally available credit along with pending terminal emissions has multiple benefits: it avoids generating FLUSH messages and maintains more credit available among active processes. When sending a primary message, a process splits its locally available credit (according to different policies detailed below) and "piggybacks" a fraction of the credit onto the message. Because the piggyback is of fixed size (since our credit representation does not grow to remain infinitely divisible), the practical cost of adding the piggyback to primary messages is trivial. However, it is possible that a process that needs to emit primary messages would run out of locally available credit. In this case, the process requests (with a secondary *BORROW* message) the allocation of supplementary credits from the controlling agent.

The controlling agent counts how many credits have been created during the execution in a counter that grows as necessary, thereby ensuring that the controlling agent will never fail at providing supplementary credits. As a consequence, more credit than is representable by the maximum value of local and message credit may be in circulation in the system. If a non-controller process receives a message containing more credit than it could accumulate in a single variable without an overflow, its local credit is set to the maximum, and all remaining credit is immediately returned to the controlling agent. For as long as a process is out of credit (e.g., the time period required for the secondary BORROW request to round-trip to the control agent), the process has to delay the emission of all primary messages, since it would otherwise carry the risk of resetting the destination process to active without holding message carried credit.

Running out of credit is a major performance hurdle and should be avoided. To reduce the likelihood of running out of credit, we devise two complementary strategies: (1) the credit division strategy that we employ operates under multiple regimes, and (2) credit borrowing is prefetched.

The minimum credit that a primary message may safely carry is 1. While this strategy reduces the attrition rate at the sender process (by leaving as much credit as possible at the source), if a message reaches a process that has little credit left (e.g., an idle process that had rid itself of all its local credit), then that process will need to borrow credit from the controlling agent and delay the next primary message. Conversely, if a process divides the credit into two halves for every message (as is customary in many CDA algorithms, including HCDA), then local credit declines very quickly (at an exponential rate) with the number of outgoing messages – leading to a high chance of the process running out of credit before it receives credit naturally through its primary message receptions.

We devise a multi-regime strategy that avoids both issues. When a process holds abundant credit (i.e., above a threshold value, C_{con}) the process employs a credit division strategy to improve the chances that destination processes may carry more message emissions without borrowing. Multiple messages may be sent simultaneously (from the view of the emitter process and independently of the port model of the network) when a task creates multiple successors at remote processes. Each individual successor task may represent an individual emission, yet all are created during the same local step. Message emissions may also appear simultaneous for a process when considering an asynchronous communication system that enqueues non-blocking emissions. Messages may be scheduled from additional tasks that are completing at the local process before the initiation of previously scheduled emissions at that same process. In both cases, instead of dividing the credit by two for every message, credit is divided uniformly among all outstanding emissions when message emissions are *simultaneous*. We maintain a counter of shares, S, which counts how many shares are known for the current credit. S is equal to the number of outgoing messages, plus one if the process remains active. Letting C_{cur} denote the current credit amount, each message receives $\lfloor C_{cur} \rfloor / S \rfloor$ credits.

When local credit drops below C_{con}, the allotment of credit per message is modified to carry a fixed amount of credit per message W_{con}. The goal is to conserve the local credit to enable the process experiencing low availability of credit to keep issuing messages with no delays, for as long as possible. Overall, the credit allocation function uses the following formula to set the credit, w_i, on an outgoing message, m_i, at a process with current credit, C_{cur}, and S shares.

$$w_i = \begin{cases} \lfloor \frac{C_{cur}}{S} \rfloor & \text{if } C_{cur} > C_{con} \\ \min(\lfloor \frac{C_{cur}}{S} \rfloor, W_{con}) & \text{if } C_{cur} \leq C_{con} \end{cases}$$

In addition, to further avoid delaying emissions, when less than C_{borrow} is available, the process proactively issues a BORROW message to replenish its credit with additional credit

from the control process. The amount of credit returned by the control process is C_{init}. In some cases, this may increase the number of secondary BORROW messages, as the process may have received credit (from primary message receptions) before reaching an indivisible credit, but the severe performance penalty resulting from delaying primary messages supports the deployment of this optimization.

V. Analysis

In this section, we compare the 4C, EDOD, HCDA, and CDA algorithms in terms of their number of control messages, which is the key parameter to assess and compare their respective overhead. We use two simple applicative kernels for this comparison: (1) the token ring, which is the archetype case study for distributed algorithms; and (2) the projection operation in $1-$dimensional trees described in Section II, which is representative of tree-based synchronous computations.

A. Token ring

The token ring is a kernel widely used to assess the performance of distributed algorithms [9], [10], [11]. Informally, it consists of several steps, with a token randomly moving from one process to another at every step, and a random number of steps. We use the following instantiation.

- The token is initially owned by process 0.
- With a fixed probability of $p < 1$, the token owner draws a process number randomly and uniformly in $[0, \mathcal{P} - 1]$ and sends a message (the token) to that process. The algorithm stops with a probability of $q = 1 - p$.

The expected number of steps (token moves) of the algorithm is $\frac{1}{q}$. At each step, the token owner performs some computation, the precise length of which is not important but is assumed to be long enough so that all control messages of the termination detection algorithm are processed before the next step begins. In other words, we can view the steps as synchronized, with the termination algorithm detecting termination (or not) at the end of each step.

The token ring mimics the termination pattern of an application that ends with a linear chain of tasks, the length of which is data dependent. Our results are shown below in Theorem 1.

Theorem 1. *The expected number of control messages of* 4C*,* EDOD*,* HCDA*, and* CDA *for the token ring is the following:*

- $\mathbb{E}(\text{4C}) \geq \frac{1}{q} \times \mathcal{P} + \mathcal{P} + o(\mathcal{P})$
- $\mathbb{E}(\text{EDOD}) \geq \frac{1}{q} \times 3\log(\mathcal{P}) + 2\mathcal{P} + o(\mathcal{P})$
- $\mathbb{E}(\text{HCDA}) = \frac{1}{q} + \frac{2}{log(C_{init})q} + \mathcal{P} + o(\mathcal{P})$
- $\mathbb{E}(\text{CDA}) \leq 2\mathcal{P}$

We see that EDOD is more efficient than 4C at each step, and that CDA is the clear winner as soon as the token circulates at least $\mathcal{P}$ times.

Proof. At each step of the token ring algorithm, the sender node makes an *active-to-idle* transition, while the receiver node is awakened by the token message and makes an *idle-to-active* transition. Because we assume the steps do not overlap, these are the only two transitions during the step, and all the other processes remain idle.

For the 4C algorithm, the sender initiates a chain of messages by notifying its parent in the control tree. There are two cases, described below.

- If the receiver is not an ancestor of the sender in the control tree, it will notify its parent, which in turn will notify its parent, thereby eventually reaching the root. If the current step is not the last step, the root will detect that the current wave has failed (because not all nodes have reported being idle) and will propagate this information down to tree to all processes via a descending wave; if this is the last step, the root will detect termination and send the final descending wave; in both cases, the cost is $\mathcal{P} - 1$ control messages.
- If the receiver is an ancestor of the sender in the control tree, the chain of messages from the sender to the root will be blocked by the receiver. But this latter event has a small probability, because there are at most $\log(\mathcal{P})$ nodes in the path from the sender to the root. Hence, the probability of the receiver belonging to that path is at most $\frac{\log(\mathcal{P})}{\mathcal{P}}$.

Altogether, the expected number of control messages per step is at least $(1-\frac{\log(\mathcal{P})}{\mathcal{P}})(\mathcal{P}-1)+\frac{\log(\mathcal{P})}{\mathcal{P}}\times 1 = \mathcal{P}+o(\mathcal{P})$. Adding the cost, $\mathcal{P} - 1$, of the final notification broadcast, we get the result for $\mathbb{E}(\text{4C})$, since the expected number of steps is $\frac{1}{q}$. For the EDOD algorithm, we have the following analysis.

- Initially, every node transitions from active to idle, either immediately or after sending the first message for the initiator, and sends a message to its parent in the control tree; therefore, there are $\mathcal{P} - 1$ messages.
- For each token message at each step, an acknowledge message is sent by the recipient to the sender. It goes through a chain of *resume* and *acknowledge* all along the unique path in the control tree connecting both nodes. The number of control messages is equal to the distance between both nodes in the control tree. The average distance between two nodes in a complete binary tree of $\mathcal{P}$ nodes is asymptotically $2\log\mathcal{P}$ [12]. As a side node, we see that this average distance is of the same order as the diameter of the tree, which can be explained by the fact that the majority of nodes are leaves of the tree (see [12] for further details).
- We have to add the *stop* messages, propagated by the sender up to the tree, which leads to $\log\mathcal{P}$ additional messages per token message. Altogether, the overhead is $3\log\mathcal{P}$ per step.

Adding the cost, $\mathcal{P} - 1$, of the final notification broadcast, we get the result for $\mathbb{E}(\text{EDOD})$. Finally, we discuss the number of control messages for the credit distribution algorithms. For HCDA, we count a message (to return the credit) every step and two messages (borrowing request and extra credit) every $\log(C_{init})$ steps, when the credit piggybacked in the primary message runs out. For CDA, this means the following.

- After the first step, process 0 (the source node) becomes idle after the token message is sent, and it transfers all its current weight, C_{init}, into the token message and has nothing to return to the controlling agent. All nodes except the source

node were active and became idle during the first step, hence they return their total weight to the controlling agent, which amounts to $\mathcal{P}-1$ control messages.
- While the token iterates during the following steps, the sender has weight, C_{init}, and transfers it into the message, and then it has zero weight and does nothing more. The recipient had weight, 0, and gets C_{init} from the message.
- Upon termination, the recipient sends its weight, C_{init}, back to the controlling agent.

Altogether, the overhead is $\mathcal{P}$ messages (out of which $\mathcal{P}-1$ are sent during the first step), hence the result for $\mathbb{E}(\text{CDA})$ when adding the cost of the last termination broadcast. An important distinction for CDA is that the total number of control messages is independent of the number of steps. □

B. Non-deterministic binary trees

We consider now the projection operation in $1-$dimensional trees described in Section II. In practice, we can model such an operation with a task graph that unfolds a binary tree, each node having two children with some probability, and being a leaf otherwise. Since an exact analysis with arbitrary task weights is out of reach, we present a simplified scenario to evaluate the average performance of the four algorithms. The simulation works as follows: first, we precompute some application trees with the following algorithm:

(1) Start with a complete tree of height $L_{min}=3$.
(2) For each leaf, at level l with probability λ^l, refine by replacing the leaf with a complete subtree—the height of which is drawn uniformly and at random between 2 and 5 (i.e., we add between 2 and 30 new nodes).
(3) Repeat the last step on all new leaves until no leaf is refined.
(4) Crop the tree if its height exceeds L_{max}.

The tasks of the tree are labeled using a breadth-first order: task 0 is at level 0, and tasks 1 and 2 are at level 1, and so on. We generated different sizes of trees using the following parameters: small trees with $\lambda=0.8$ and $L_{max}=30$, medium trees with $\lambda=0.9$ and $L_{max}=50$, and big trees with $\lambda=0.93$ and $L_{max}=60$.

For the simulation, we consider that all tasks at a given level, l, are processed at time, l. We have two different mapping strategies for mapping tasks to processes: (1) a round-robin mapping, where task x goes to process $x \bmod \mathcal{P}$; and (2) a random mapping, where task x goes to a process uniformly drawn in $[0, \mathcal{P}-1]$.

We compute the messages sent by the application at each step (all tasks at a level in the tree), and determine whether the processes become active or idle at the end of the step. When a process was active and is again active at the end of the step, we model the inherent distributed aspect of the algorithms using three different models:

• **$S_{instant}$:** The node does not transition to idle during the step, it remains active throughout. This corresponds to the case where computations and communications are instantaneous, thus a node knows in advance whether it will stay active or not.

• **S_{local}:** The node transitions to idle before returning to active, unless there is a message to itself. This corresponds to the case where communications are very slow compared to computations, all messages are received at the end of the step, so a process transitions to idle because it cannot know in in advance whether it will stay active or not.

• **S_{load}:** The node transitions to idle before returning to active only if it has no message for itself and if its load is smaller than all the loads of the nodes that send a message to it: this is because in that case, it terminates computing before receiving any load from the other guys. This corresponds to the case where computations are long and messages takes very short time. We define the load to be equal to the number of messages received at the previous step (each of them implying the execution of a task, this corresponds to assuming that all tasks have the same weight).

To compare the performance of CDA to other algorithms, we compute the number of control messages sent by each algorithm, as detailed below.

- **HCDA** and **CDA**: all messages carry credits, so there is no control message – except when one process becomes idle and needs to return its credit to the controlling agent (flush), or when it does not have credit anymore and needs to send a message to the controlling agent to continue (borrow). Each time we detect that a process needs to flush or borrow, we add one control message. Otherwise, when processes transition from idle to active or from active to idle, we do not count anything, as these algorithms do not send messages for simple transitions.
- **4C**: once the list of messages (sent during a step) is computed, we go through the list of all processes in descending order. If a process becomes idle, we check if it belongs to the wave. If it does not, it is added to the wave; if the process has children, they also belong to the wave. By going through the processes in descending order, we ensure the wave goes as high as possible in the control tree. Each time the root belongs to the wave, we account for $2(\mathcal{P}-1)$ messages ($2\times$ the number of edges in the control tree).
- **EDOD**: each time a process, p_i, transitions from idle to active, it means that it received messages from a set of processes, S. We then compute the union of all paths from p_i to each one of the processes in the set S. Finally, we sum the number of edges in that union of paths, which accounts for the number of control messages sent at this step by process p_i. When a process, p_i, transitions from active to idle, we check that its whole subtree is composed of idle processes at the end of the step. In that case, we account for one control message that goes up in the control tree; there are n messages total, where n is the size of the subtree, because each node of the subtree is the root of an idle subtree itself.

Figures 2 to 5 present the number of control messages for all algorithms. We had to use a logarithmic scale on the Y-axis to report a range of different numbers. The data is presented for a wide range of tree sizes, ranging from small (47 tasks in Figure 2) and medium (397 tasks in Figure 3)

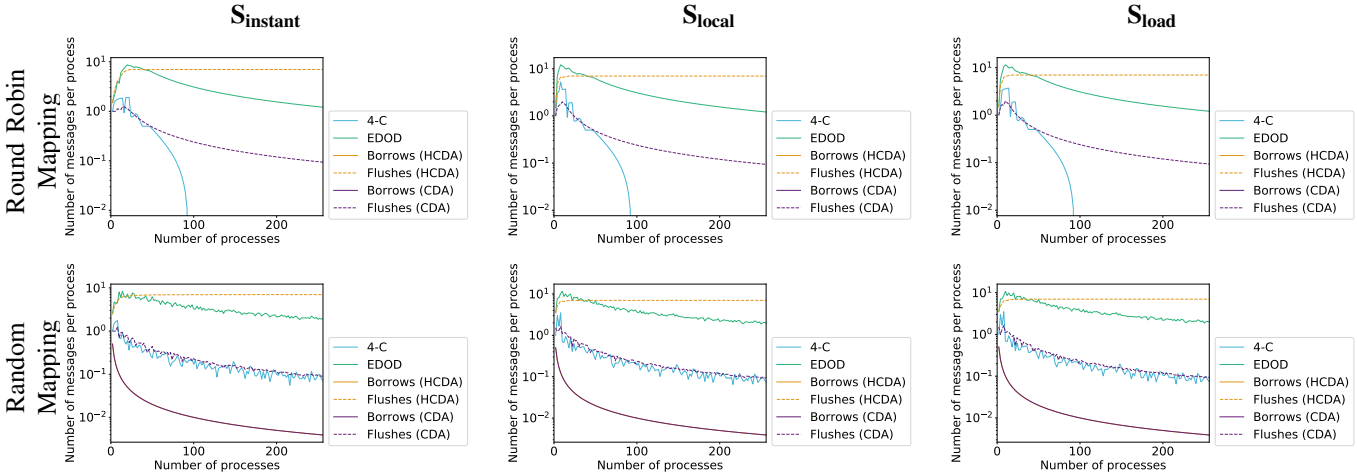

Figure 2: Number of control messages per process for all algorithms (4C, EDOD, HCDA and CDA) for a 47-task tree. The first row uses a round-robin mapping while the second row uses a random mapping.

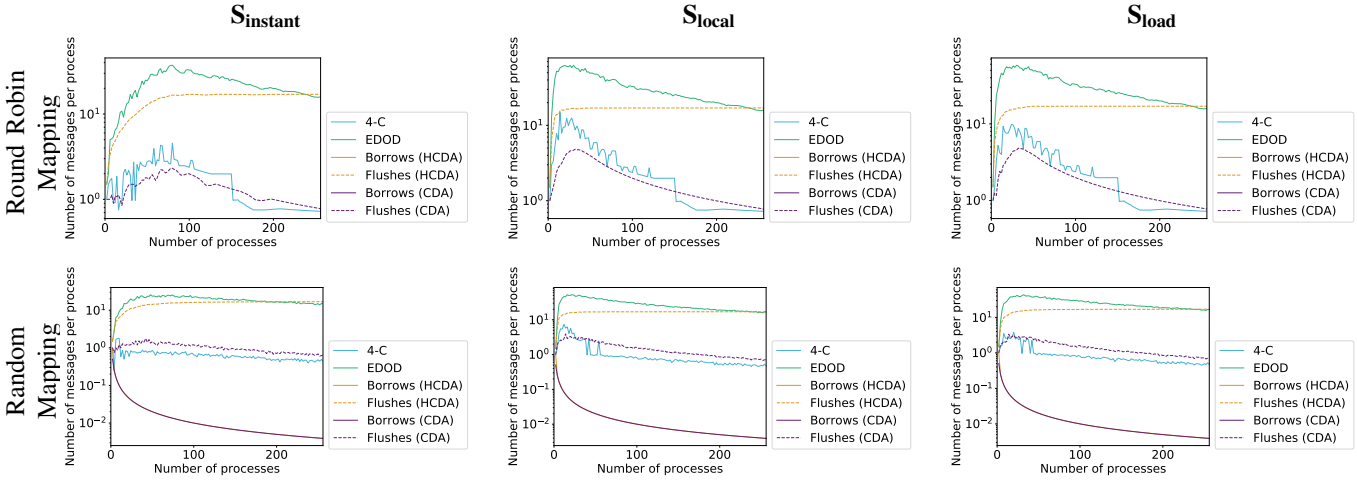

Figure 3: Number of control messages per process for all algorithms (4C, EDOD, HCDA and CDA) for a 397-task tree. The first row uses a round-robin mapping while the second row uses a random mapping.

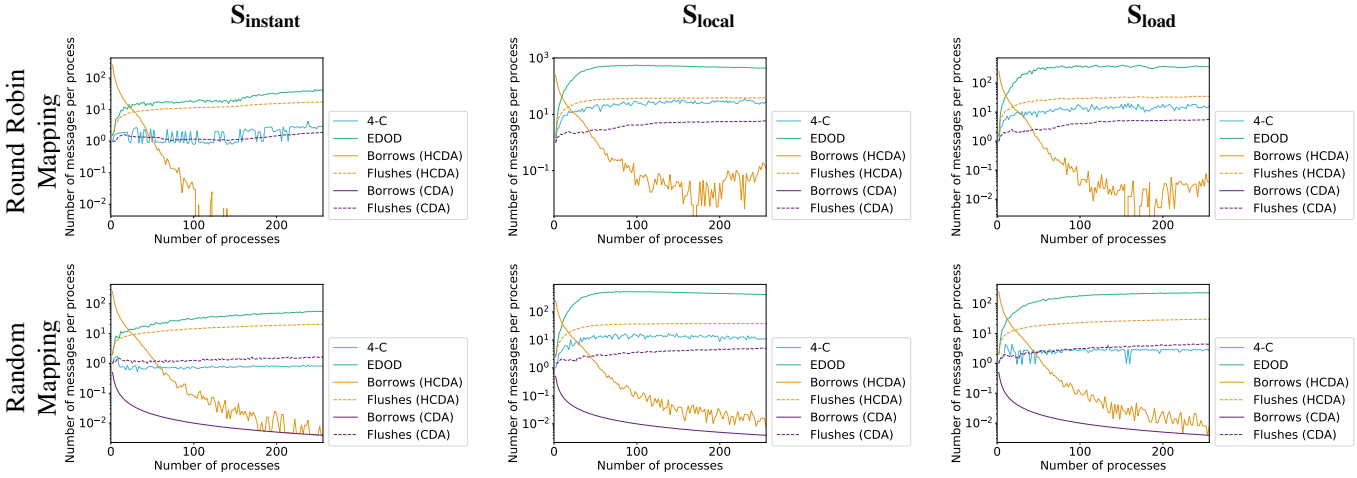

Figure 4: Number of control messages per process for all algorithms (4C, EDOD, HCDA and CDA) for a $17,797$-task tree. The first row uses a round-robin mapping while the second row uses a random mapping.

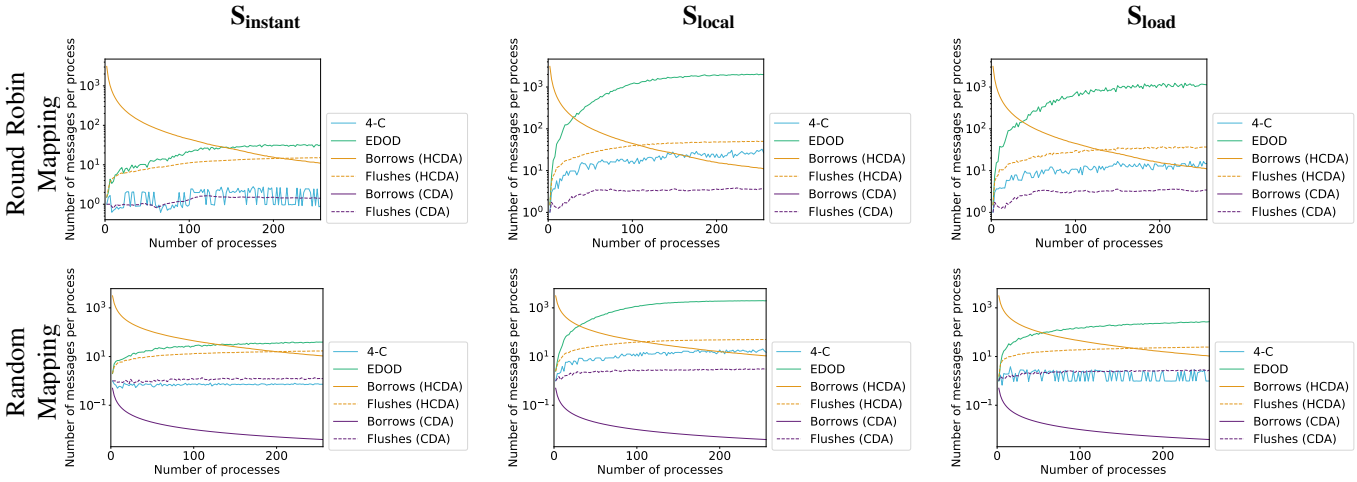

Figure 5: Number of control messages per process for all algorithms (4C, EDOD, HCDA and CDA) for a $202,007$-task tree. The first row uses a round-robin mapping while the second row uses a random mapping.

to large ($17,797$ tasks in Figure 4) and very large ($202,007$ tasks in Figure 5), using an initial credit, $C_{init} = 2^{32}$. First, all the simulations show that CDA dramatically outperforms HCDA. Interestingly, the only occurrences of *BORROWs* for CDA are when the mapping is random and there are only a few processes. In this case, the processes may receive a lot of messages, thus a lot of tasks to execute, and thus a higher number of messages to send afterward. With the credit reducing quickly at the beginning because there is a good probability that a process is idle in the first steps, and there may be too many messages to send compared to the credit. Still, the number of *BORROWs* is – on average – null, especially when using round-robin mapping. The only overhead in terms of messages added by CDA comes from the number of flushes (when a process becomes idle and has no message to send). When using round-robin mapping, the number of flushes per process is less than the number of control messages sent by the 4C algorithm (on average for the first figure). However, when we set the mapping to be random, 4C proves to be more efficient than CDA when $\mathcal{P} > 100$. Between random and round-robin mapping, the number of control messages for CDA does not change much, whereas random mapping drastically reduces the number of control messages for 4C.

Overall, we expect CDA to send less messages than 4C, in particular when the number of processes increases. Looking at the top-right plots in each figure, where the model is $\mathbf{S_{load}}$ and the mapping is round-robin (achieving more load balance than random), the number of flushes per process tends to stay constant when the number of processes increases, whereas 4C produces more messages.

Table I provides the average ratio, over the three models, of the number of control messages generated by 4C, EDOD, and HCDA over that of CDA, for the four tree sizes. For the round-round mapping, the best competitor is 4C which sends three times fewer messages for 47-task trees but three and a half times more for $202,007$-task trees. For the random mapping, 4C is also the best competitor, with a 20% gain for 47-task trees but twice more messages for $202,007$-task trees. For both mapping, EDOD and HCDA generate an order of magnitude more messages than CDA. Altogether, for large trees, CDA succeeds in dramatically reducing the total number of control messages in comparison with the other three algorithms 4C, EDOD, and HCDA.

VI. Related work

[13] proposes a method to precisely define the metrics of efficiency for distributed termination detection. We leverage this method in our analysis.

Termination detection has been studied extensively from the theoretical perspective: [14] demonstrates that different classes of detectors are equivalent through automatic transformations; see Ch. 6 of [1] and Ch. 9 of [15].

Wave termination detection algorithms include [16], based on distributed snapshots, and [17], designed for asynchronous wide-area networks by combining a reduction tree with a logical ring. Delay optimal algorithms include [18] and [6], and we compare one that is representative to this work. Weight throwing, or distributed credit algorithms, have been extensively studied theoretically: [19] proposes to use them to implement garbage collection mechanisms; [20] introduces the Doomsday termination detection protocol that deals with migrating tasks; [21] uses a mobile agent to count the weight remaining in the system; [22] and [23] consider the particular case of mobile networks; and [24] considers resilient approaches to these algorithms. A recent work [25] introduces resilient optimistic termination.

Few works compare, experimentally or practically, the different algorithms to evaluate the behavior in average or real-world conditions. In [26], this comparison is conducted over a simple benchmark consisting of 100 randomly generated nested graphs of tasks.

Tree size	4C round-robin	4C random	EDOD round-robin	EDOD random	HCDA round-robin	HCDA random
47	0.32	0.83	12.08	16.07	37.68	38.28
397	1.28	0.65	17.09	18.97	11.65	14.36
17,797	3.34	1.60	68.06	64.56	12.70	12.66
202,007	3.52	1.98	201.68	176.38	95.40	87.98

Table I: Average ratio of the number of control messages generated by 4C, EDOD, and HCDA over that of CDA, for the round-robin and random mappings, and for all tree sizes.

VII. Conclusion

This paper revisits distributed termination detection algorithms in the context of HPC applications, motivated by the need to efficiently detect termination of work flows for which the total number of tasks are data-dependent and, hence, not known until during execution. We introduce an efficient variant, CDA, of the credit distribution algorithm, and compare it to the initial credit distribution algorithm, HCDA, and to two other termination detection algorithms, 4C and EDOD. We analyze each algorithm for simplified task-based kernels and show the superiority of CDA in terms of the number of control messages.

Future work will be devoted to providing a highly tuned implementation of each termination detection algorithm within the task based runtime system PaRSEC [27], and to compare their performance for a variety of benchmarks reflecting scientific applications that exhibit dynamic behaviors. This will enable us to quantify the overhead of each algorithm in terms of absolute application performance, not just in terms of the number of control messages that are generated.

Acknowledgements: This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was supported partly by the NSF project #1450300.

References

[1] W. Fokkink, *Distributed Algorithms: An Intuitive Approach*. The MIT Press, 2013.

[2] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Inf. Process. Lett.*, vol. 11, no. 1, pp. 1–4, 1980.

[3] N. Francez, "On achieving distributed termination," in *Proc. Int. Symp. Semantics of Concurrent Computation*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 70. Springer, 1979, pp. 300–315.

[4] S. T. Huang, "Detecting termination of distributed computations by external agents," in *ICPP*. Pennsylvania State University Press, 1989, pp. 79–84.

[5] F. Mattern, "Global quiescence detection based on credit distribution and recovery," *Inf. Process. Lett.*, vol. 30, no. 4, pp. 195–200, 1989.

[6] N. R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," *J. Parallel Distributed Computing*, vol. 67, no. 10, pp. 1047 – 1066, 2007.

[7] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, J. Pei, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, T. Yanai, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Scientific Computing*, vol. 38, no. 5, 2016.

[9] J. Misra, "Detecting Termination of Distributed Computations Using Markers," in *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, ser. PODC '83. ACM, 1983, pp. 290–294.

[8] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[10] A. J. Martin, "Distributed mutual exclusion on a ring of processes," *Science of Computer Programming*, vol. 5, pp. 265–276, 1985.

[11] X. Défago, A. Schiper, and P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy And Survey," *ACM Computing Surveys*, vol. 36, p. 2004, 2003.

[12] B. Parhami, "Exact formulas for the average internode distance in mesh and binary tree networks," *Computer Science and Information Technology,*, vol. 1, pp. 165–168, 2013.

[13] Y. Tseng and R. F. DeMara, "Communication pattern based methodology for performance analysis of termination detection schemes," in *IPDPS*. IEEE, 2002.

[14] S. Peri and N. Mittal, "Improving the efficacy of a termination detection algorithm," *J. Inf. Sci. Eng.*, vol. 24, no. 1, pp. 159–174, 2008.

[15] S. Ghosh, *Distributed Systems An Algorithmic Approach, 2nd Edition*. Chapman & Hall/CRC, Computer & Information Science Series, 2015.

[16] S.-T. Huang, "Termination detection by using distributed snapshots," *Information Processing Letters*, vol. 32, no. 3, pp. 113 – 119, 1989.

[17] X. Wang and J. Mayo, "A general model for detecting distributed termination in dynamic systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 84–.

[18] N. Mittal, S. Venkatesan, and S. Peri, "Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies," in *Distributed Computing*, R. Guerraoui, Ed. Berlin, Heidelberg: Springer, 2004, pp. 290–304.

[19] S. M. Blackburn, J. E. B. Moss, R. L. Hudson, R. Morrison, D. S. Munro, and J. N. Zigman, "Starting with termination: A methodology for building distributed garbage collection algorithms," in *24th Australasian Computer Science Conference (ACSC 2001), 29 January - 1 February 2001, Gold Coast, Queensland, Australia*, 2001, pp. 20–28.

[20] M. J. Livesey, R. Morrison, and D. S. Munro, "The doomsday distributed termination detection protocol," *Distributed Computing*, vol. 19, no. 5, pp. 419–431, Apr 2007.

[21] N. Garanina and E. Bodin, "Distributed termination detection by counting agent," *CEUR Workshop Proceedings*, vol. 1269, pp. 69–79, 01 2014.

[22] R. Mishra and P. Saini, "A weight throwing and diffusing computation based approach for termination detection in manets," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2016, pp. 50–55.

[23] S. De, M. Sameeruddin, V. Sharma, N. Nandi, and H. Dutta, "A new termination detection protocol for mobile distributed systems," in *10th International Conference on Information Technology (ICIT 2007)*, Dec 2007, pp. 148–150.

[24] T. Tseng, "Detecting termination by weight-throwing in a faulty distributed system," *Journal of Parallel and Distributed Computing*, vol. 25, no. 1, pp. 7 – 15, 1995.

[25] S. S. Hamouda and J. Milthorpe, "Resilient optimistic termination detection for the async-finish model," in *High Performance Computing (LNCS 11501)*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Springer, 2019.

[26] R. F. DeMara, Y. Tseng, and A. Ejnioui, "Tiered algorithm for distributed process quiescence and termination detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 11, pp. 1529–1538, 2007.

[27] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.