Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander, Phil Miller, Laxmikant V. Kale
University of Illinois Urbana-Champaign
{jliffl2,mille121,kale}@illinois.edu

Abstract

Termination detection is relevant for signaling completion (all processors are idle and no messages are in flight) of many operations in distributed systems, including work stealing algorithms, dynamic data exchange, and dynamically structured computations. In the face of growing supercomputers with increasing likelihood that each job may encounter faults, it is important for high-performance computing applications that rely on termination detection that such an algorithm be able to tolerate the inevitable faults. We provide a trio of new practical fault tolerance schemes for a standard approach to termination detection that are easy to implement, present low overhead in both theory and practice, and have scalable costs when recovering from faults. These schemes tolerate all single-process faults, and are probabilistically tolerant of faults affecting multiple processes. We combine the theoretical failure probabilities we can calculate for each algorithm with historical fault records from real machines to show that these algorithms have excellent overall survivability.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel & distributed programming; D.4.5 [*Reliability*]: Fault-tolerance

Keywords Termination Detection, Fault Tolerance

1. Introduction

As parallel programs scale to larger systems, the occurrence of faults becomes increasingly likely to impact their execution [9]. At the same time, the popularity of distributed parallel programming systems that implement high degrees of dynamic behavior, such as asynchronous tasks [1], work stealing [4, 11, 12], and message-driven execution [8, 18, 19], are increasing. Unlike in bulk synchronous parallel programs, and even in dynamic data exchanges within BSP programs [7], there is often no clear global indication of when some particular distributed computation is complete. Thus, they instead rely on *termination detection* algorithms to provide that indication.

Termination in a distributed system is the state in which all processes are idle, and no message is in flight that may cause a process to become active [3]. There are many different approaches to termination detection (TD), which have been surveyed by Mattern [13]. Several researchers have constructed fault-tolerant algorithms for TD [10, 15, 16]. The past work in the area of fault-tolerant TD has focused on the problems of arbitrary failures in general distributed systems. These previous approaches handle failures of nearly the entire system, but at the cost of substantial complexity and non-scalable operations when faults occur. For instance, one algorithm routes and replicates all information used in termination detection through a constant number of processes, creating a sequential bottleneck that will grow with the system and problem sizes [16].

We approach the problem of fault tolerance (FT) from a different perspective: we should expect to experience faults, and pay only scalable and local costs to recover from faults that are likely to occur and unlikely to necessitate job termination. Specifically, in HPC systems, a large fraction of faults affect only a single node, and the likelihood of a fault decreases with the number of nodes it affects [14]. Thus, we explore several schemes to make TD tolerant of all single-process failures, and probabilistically tolerant of larger failures. The cost of recovery in our algorithm is very low, scalable relative to application behavior, and local to the communication neighborhood of the failed process. For applications whose behavior is scale-invariant [6], our algorithm is scale-invariant as well. We also show through experiments that our overhead on fault-free execution is minimal.

The primary contributions of this paper are as follows:

- We describe a new theoretical metric for TD, *process fanout*, that is used in analyzing our algorithms (§ 5.2).
- We characterize the (in)applicability of various fault-tolerant TD mechanisms to HPC applications (§ 3).
- We describe a series of three FT termination detection mechanisms (INDEP, RELLAZY, & RELEAGER) that present low theoretical overhead (both process-optimal and equal to or better than the cost of the message-optimal termination detector itself) on fault-free execution and do not impose non-scalable costs when recovering from faults (§ 5, § 6).
- We relate our algorithms' failure probabilities to fault data from real systems, to demonstrate their high practical survivability (§ 5.4, § 6.4, Table 1).
- We provide empirical measurements showing that the overhead costs of these schemes in a highly-scalable parallel runtime are low in practice (§ 7).

2. Background: Parental Responsibility Termination Detection

In constructing our fault tolerant termination detection algorithms, we build on the seminal work of Dijkstra and Scholten in defining the *parental responsibility* approach to termination detection [3, 13]. Our algorithms are constructed in terms of extensions to their original scheme. In this section, we introduce a concrete implemen-

PPoPP'13, February 23–27, 2013, Shenzhen, China.

tation of their algorithm, the invariants that they proved it obeys, and how those invariants provide for its correctness. We omit proofs of these invariants, since they can be found in their paper.

The computation is distributed among *processes* in a system. It is assumed to start at a particular *root process* which will eventually signal termination. All other processes are initially idle, or passive, and cannot be activated except by receiving a message from an already active process. This structure is generally known as a diffusing computation [3].

The general intuition of parental responsibility termination detection is that every message the application sends (also referred to as basic or primary messages) will eventually be *acknowledged*, and when all messages have been acknowledged, then termination is detected. The key to detecting termination using message acknowledgment is that some acknowledgments cannot be sent immediately, but must be delayed until the recipient can be sure that all work its messages have initiated in the system is complete.

To accomplish this, some processes are characterized as *engaged*, which means it or some process that was transitively activated by it is actively working, and all other processes are unengaged. The root is initially engaged, and all others are initially unengaged. An unengaged process becomes engaged upon receiving a message. An engaged process becomes unengaged when it has processed and acknowledged all messages sent to it and all messages it has sent have been acknowledged.

Every message carries an indication of its origin, denoting for the recipient who it must acknowledge, and when an unengaged process receives a message, that sender process becomes its *parent*:

```
def gotMsg(Endpoint predecessor):
  if (cornet.size() == 0):
    parent = predecessor;
    cornet.insertParent(parent, 1);
  else:
    cornet[predecessor].acksOwed++;
```

Dijkstra and Scholten describe storing the acknowledgments a process must send in a structure called a *cornet*, which distinguishes the first element placed in it to be removed last, and all other entries can be accessed or removed arbitrarily. In other words, it is 'very first in, very last out' and otherwise unordered. The subset of processes that are engaged form a directed tree determined by their parents, which is called the *engagement tree*. We describe an engaged process with parent p as *engaged to* p.

As processes consume messages that they receive, they may send messages to other process. That creates a debt of acknowledgments that must be repaid before the process can disengage, stored in D, which is initially zero:

```
def willSendTo(Endpoint successor):
  D++;
```

When these messages are acknowledged, the debt is reduced:

```
def gotAck(Endpoint successor, unsigned int count):
  D -= count;
  tryDisengage();
```

Once a process consumes a message, the process gains credit which can be used to acknowledge a message, stored in C, which is initially zero:

```
def processedMsg():
  C++;
  tryAck();
  tryDisengage();
```

With credit in hand, it is possible that the process may safely acknowledge some messages. The process checks whether it has consumed enough messages to acknowledge some source other than its parent (chosen arbitrarily), and if so, transmits that acknowledgment and reduces its available credit:

```
def tryAck():
  if (cornet.size() <= 1): return;
  Endpoint predecessor = cornet.chooseNonParent();
  int a = cornet[predecessor].acksOwed;
  if (C >= a):
    C -= a;
    @predecessor { gotAck(self, a) };
    cornet.remove(predecessor);
```

We use the notation `@proc { foo(); }` to indicate sending a message to process `proc` asking it to execute the enclosed code. These messages are assumed to be processed between basic messages, and not in a preemptive manner, such as in an interrupt.

When a process has no one left to acknowledge but its parent, it may try to disengage:

```
def tryDisengage():
  if (cornet.hasNonParent()) return;
  if (D == 0 && C == 1):
    if (isRoot()) rootTerminated();
    else:
      @parent { gotAck(self, 1); }
      cornet.clearParent();
    C = 0;
```

In order to do so, all messages it sent must have been acknowledged ($D = 0$), and the only acknowledgment debt it still owes must be to its parent ($C = 1$).

INVARIANT 1. $C_p \geq 0 \land D_p \geq 0$

Both accounting variables are non-negative on all processes.

INVARIANT 2. *Process p being engaged is equivalent to* $\sum_{pred \in \mathrm{cornet}_p} \mathrm{cornet[pred].acksOwed} > 0$

INVARIANT 3. $D_p > 0 \rightarrow$ *process p is engaged.*

A process must be engaged in order to send messages, and thus increase D above zero.

During execution, we denote the number of messages the application sends by M. The Dijkstra-Scholten algorithm sends an additional $\mathcal{O}(M)$ messages to accomplish its purpose.

The correctness of this TD algorithm is defined by two properties: it does not detect termination while any process is still engaged, and it detects termination when all processes have disengaged. For purposes of fault tolerance, we generalize this correctness to mean that the algorithm does not detect termination while any process is still engaged, and it either detects termination when all surviving processes have disengaged or reports a fatal error when it can no longer determine when that has occurred. When faults occur, it is still the application's responsibility to determine what work was lost in the fault and how to address that loss.

3. Related Work

The goal of the present effort is to describe an algorithm that is scalable – no single process or narrow subset of processes should be burdened with work that grows disproportionately from the effort it expends in executing the client application. Existing algorithms do not satisfy these desires, either during fault-free execution or during fault recovery.

Venkatesan's algorithm designates a number of 'leader' processes that are responsible for declaring termination [16]. To tolerate k-process faults, it must have at least $k+1$ leaders. Each of those leaders receives termination detection signals from all of the processes in the system, and must store and simulate the state of all of those processes to track when they have terminated. This presents a clear sequential bottleneck, and a potential memory overload, that is impractical in a high performance computing environment.

The protocol described by Lai and Wu [10] avoids all overhead during fault-free execution. However, in the event of a fault, every surviving process communicates directly with a designated root process. While the total number of messages necessary does not scale beyond the system's scale, the root process becomes a sequential bottleneck that will take $\mathcal{O}(N)$ time to receive these messages, process them, and respond. As systems scale up, this will lead to recovery taking longer than the time between failures, and thus ultimately does not achieve its goal.

Much recent research in fault-tolerant parallel computing has focused on *algorithmic fault tolerance* [6]. This approach does not generically try to make entire parallel applications fault-tolerant, but instead addresses itself to single component algorithms and libraries, with the expectation that these can later be composed to construct applications that derive their fault tolerance from that of the underlying pieces. Algorithmic fault tolerance has gained particular traction in numerical linear algebra [2], where the problems have structural characteristics that provide low-cost recovery mechanisms. Our work can be viewed as providing a fault-tolerant version of a scalable termination detection component, which can be readily integrated with other resilient components.

One particular environment where the need for fault-tolerant termination detection appears is work stealing with idempotent updates to global data [12]. That paper describes a mechanism to write distributed work stealing code that reads and writes global array structures, which are commonly used in computational chemistry applications. The work stealing mechanism provides load balance across the system, and relies on a termination detector to determine when there is no work left in the system. The tasks are arranged and programmed such that their execution is idempotent with respect to the global data. Thus, when a process fails, any potentially affected tasks are simply re-executed. The work stealing machinery maintains sufficient records to identify the subtrees that were stolen by a failed process, and can restart them. The system described in that paper is implemented in terms of a termination detector that is not itself fault tolerant. Thus, while it can be shown to survive simulated faults, it was not suitable for production use. We provide the necessary implementation of a fault-tolerant termination detector, and measure its overhead using a similar set of benchmarks.

4. System Model

In describing a fault-tolerant termination detection scheme, we make a number of assumptions about the parallel computer and interconnection network that it will run on. We do not rely on synchronized delivery of messages between sender and receiver, but instead let processes transmit message asynchronously and obliviously. The network may then deliver these messages in any order. We assume the network itself does not fail, or that such failures are accepted as leading to complete job failure.

We require a sort of 'network send fence' that permits us to conclude that a particular message has been successfully transmitted (but not necessarily received or processed) before sending another. This is a weaker assumption than some other schemes that depend on transmitting pairs of messages simultaneously [16]. However, a network satisfying such an assumption meets our own as well.

We assume that failures are *fail-stop* - i.e. failed processes do not recover [5], and do not behave maliciously (no byzantine failures). We make no assumption regarding process replacement after failure, since a fresh process is free to engage just like the initial set of processes do, by receiving a message.

In the event of a failure, we require two things of the underlying system. The first is that communication partners of failed processes receive notifications of the failure, so that they can react accordingly. In a closely-coupled HPC environment, it is reasonable to assume that an out-of-band mechanism such as the system's resource manager can provide this facility. The second requirement is a primitive to ensure that all messages sent by a failed process to a particular recipient have been received. Venkatasan refers to this facility as *fail-flush* [16].

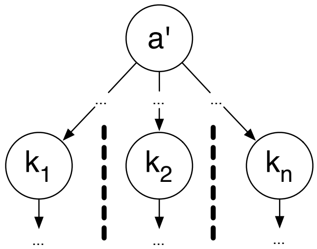


Figure 1: The INDEP protocol (§ 5) handles independent failures: k_1, $k_2 \ldots k_n$ can all fail simultaneously as long as they do not communicate.

To avoid undue complication of the descriptions, we do not explicitly address the failure of the root process. This could be mitigated in the protocol by electing or designating a backup process to take its place, which can be done with low execution-time cost but substantial implementation complexity. It would also be possible for the (assumed reliable) environment to take on the root's responsibilities, either initially, or in the event of its failure. As a single process in a prospectively large system, randomly distributed faults are unlikely to include the root process. For it to be worthwhile for our protocols to handle root failure, any parallel application or programming environment using our scheme must also be tolerant of the root failing, which imposes a similar complexity burden on them as well. Given our starting assumption that some unrecoverable faults are acceptable, as long as they are rapidly and reliably reported, we thus do not consider this vulnerability to be of fundamental importance.

5. INDEP: Tolerating Independent Failures

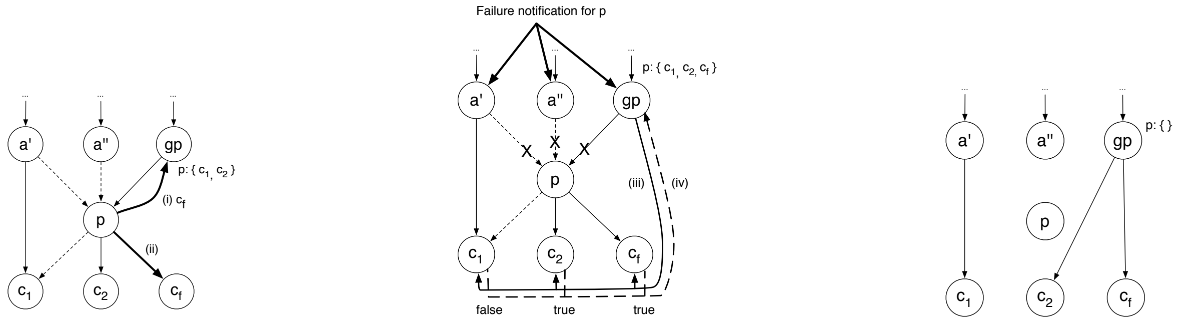
Our general approach to making these termination detection algorithms fault tolerant is to make potential parents of a failed process responsible for retaining and exploiting the information necessary for recovery. Who these children are and how they relate to a failed process is communicated through additional control signals. When processes fail, their parents delay termination until recovery has progressed far enough to take stock of their children and reattach them to the engagement tree as appropriate. If that is found to be impossible, our protocols reliably and rapidly report overall failure.

We begin by describing INDEP, a protocol for recovery from the failure of any single process or sets of independent processes that are not related by direct communication, as shown in Figure 1. These cases are always recoverable, and the completion of the recovery process is clearly delineated by the receipt of particular control signals between surviving processes.

The intuition behind INDEP is that any time a process p is about to send a message to a new recipient p_c, p ensures that its grandparent p_p will receive a message informing it of the possibility that p_c was a child of p should p fail. When a failure of p occurs, p_p will exchange messages leading it to adopt the children p_c as its own.

5.1 Modifications to Fault-Free Execution

When sending a message, each process records how many unacknowledged messages it has sent to each destination. For messages



(a) Extra messages sent during forward execution: p sends a `potentialGrandChild` message (i) before the first application message it sends to a new recipient (ii).

(b) The recovery process: processes that sent messages to the failed process p are notified. The parent, gp, contacts its potential grandchildren to see which were orphaned by the failure and should be adopted as its children.

(c) The state after recovery: All processes have written off their debts to or from p. Depending on the responses from (c_1, c_2, and c_f), gp will modify its deficit by the count of its new children so it appropriately waits for them to finish.

Figure 2: The INDEP fault tolerance protocol.

that might engage a child, it also notifies its parent of the new prospective child (Figure 2a). This is where we depend on a network fence – if the child process receives the application message, that child's potential grandparent process must have been sent a notice regarding its presence.

```
def willSendTo(Endpoint successor):
  D++;
  successors[successor].acksOwed++;
  if (successors[successor].acksOwed == 1):
    @parent { potentialGrandchild(self, successor) };
    requestFailureNotice(successor);
    sendFence();
```

Because of the potential fault recovery responsibility that comes with sending a message to a process, the termination detector must request failure notifications regarding that process as well.

During forward execution, the grandparent processes record these notifications on a per-child basis:

```
def potentialGrandchild(Endpoint child, Endpoint gc):
  successors[child].potentialChildren.insert(gc);
```

During fault-free operation, the grandparents' records are never consulted. When process p receives an acknowledgment from process q, p's corresponding debt counter for messages to q is decremented. When q's debt goes to 0, indicating disengagement, its failures are no longer of interest, and any potential grandchildren it might have introduced are forgotten, since they are guaranteed to have previously disengaged from the child process:

```
def gotAck(Endpoint successor, unsigned int count):
  D −= count;
  successors[successor].acksOwed −= count;
  if (successors[successor].acksOwed == 0):
    dropFailureNotice(successor);
    successors.erase(successor);
  tryDisengage();
```

We also introduce a counter of pending recovery events named `pendingRecoveryEvents`, which is initialized to zero. The condition guarding disengagement must additionally check that this is zero. Its use is described below.

LEMMA 5.1. *In the absence of faults, termination is detected correctly.*

Proof. The additional messages and data structures are irrelevant, with the exception of the pending recovery event counter. This is initialized to zero and is never changed during fault-free execution. Thus, when termination would otherwise occur, the added condition that this counter is zero is satisfied. Therefore, we conclude that there is no change in the fault-free operation of the fault-tolerant variant of our termination detector, and the original algorithm's proof of correctness continues to hold. □

5.2 Fanout Bound

For every process, there is some number of other processes that it will send messages to over the course the computation. In scalable applications, this number is typically limited, and should not grow with the size of the computer system or the problem being solved. In particular, within a given phase or step of a computation that is run with termination detection, each process is only likely to communicate with a reasonably small subset of the system, and we refer to the size of the largest such subset as the application's *fanout* during that phase or step. For example, in preparing the data exchange necessary to obtain column data in sparse matrix-vector multiplication, this would be the number of processors having entries on that column [7].

We use the fanout, denoted f, to describe the bounds on our algorithms' costs. Note that $f \in \mathcal{O}(M)$ – it can never exceed the number of messages the application itself sends. In practice, it may be much lower. Where that is the case, we will see that our FT overheads scale more slowly than the cost of TD itself. The fanout is also bounded by the number of processes – it is impossible for any one process to send to more destinations than exist in the system.

Fault-free execution of INDEP sends additional control messages corresponding to the cumulative fanout of the system, distributed according to the engagement structure that the application's messages create. Each process stores its potential grandchildren in a structure with entries corresponding to its own fanout, and each entry lists grandchildren according to that child's fanout. Thus, it will occupy space that is $\mathcal{O}(f^2)$ in the worst case. These structures can be implemented as hash tables, allowing constant-time lookup.

5.3 Fault Recovery

When a fault occurs that kills process p_f, the processes that have interacted with it cooperate to use the extra information they have stored to effectively ‘contract’ around its vertex in the engagement tree (Figure 2b). All of the processes that sent primary messages to p_f will receive notifications from the environment, indicating the failure and complete flushing of messages. These processes can all write off any debt p_f owed them, and ignore further acknowledgments from p_f:

```
def failureHappened(Endpoint failedProcess):
  if (successors.contains(failedProcess)):
    D -= successors[failedProcess].acksOwed;
    foreach (Endpoint grandchild
               : successors[failedProcess].potentialChildren):
      pendingRecoveryEvents++;
      willSendTo(grandchild);
      @grandchild { tryEngageGrandchild(self, failedProcess) };
    successors.erase(failedProcess);
```

One of the processes that sent p_f messages will discover that it was the failed process’s parent, and is thus responsible for waiting on the termination of its children. The recovery counter is increased by the number of potential grandchildren that must be dealt with (zero for non-parent processes). The parent p_g informs each of them that p_f has died and tries to engage them.

Each of p_f’s potential child processes will receive a message from the failed process’s parent p_g notifying it of the failure. All of these processes write off any acknowledgment debt they owed p_f. If p_f was their parent, they then adopt p_g as their parent, and replace their debt to p_f with a parental debt of 1 to p_g. They then each send a message to p_g indicating what new debt, if any, they owe it, and any new potential grandchildren that p_g must be aware of as a result of having adopted an additional child.

```
def tryEngageGrandchild(Endpoint gp, Endpoint failedParent):
  if (cornet.contains(failedParent)):
    C -= cornet[failedParent].acksOwed;
    cornet.erase(failedParent);

  if (failedParent == parent):
    parent = gp;
    C++;
    cornet.insertParent(parent, 1);

    @gp { replyToGrandparent(self, 1, keys(successors)) };
    tryDisengage();
  else:
    // Potential grandparent expects a response, even if we don't
         engage it
    @gp { replyToGrandparent(self, 0, {}) };
```

When p_g receives these responses, it decrements the recovery counter, notes any additional debt new children owe it, and records their associated potential grandchildren:

```
def replyToGrandparent(Endpoint child, unsigned int debt,
                       set<Endpoint> children):
  if (debt != 0):
    D += debt;
    successors[child].acksOwed += debt;
    successors[child].potentialChildren.insert(children);
  pendingRecoveryEvents--;
  gotAck(child, 1);
```

As each potential child’s response is received, there is the possibility that termination has been reached, and so we test for it.

Note that there is a potential race between an affirmative response from a former child of p_f to p_g and a subsequent disengagement acknowledgment message from that child to p_g. Without treating each outstanding potential child as a message send, this race may cause the value of D on p_g to reach zero unexpectedly, or even become negative. By incrementing and decrementing D along with the recovery counter, we ensure that Dijkstra’s $D > 0$ invariant is maintained.

LEMMA 5.2. *In the presence of faults, this system does not report termination when it has not occurred.*

Proof. Consider a process p_f that fails while it is engaged to parent p_p. After the failure, p_f is effectively passive, but any processes that received messages from it may still be active. When the failure notification reaches p_p, it is aware of all such potential recipients, because of the send fence before the first message to each new recipient and the fail-flush before the failure notification. If p_f had no potential children, then its subtree of the engagement tree is complete at its failure, and p_p can disengage when its other debts are repaid. If p_f had children, p_p is prevented from disengaging until each of those children responds by the non-zero recovery counter. There are four possibilities for each child’s state when the message from p_p arrives:

1. It is passive and not engaged: it responds indicating that it presents no cause for p_p to wait before disengaging.
2. Engaged to some other process: it responds indicating that another process is already prevented from disengaging until it does. Thus, p_p does not have to wait on it before disengaging.
3. Engaged to p_f: it responds that p_p is its new parent, which causes p_p to increase its outstanding debt by 1. This prevents p_p from disengaging until the newly acquired child disengages.
4. Failed: p_p is notified of the failure after requesting failure notices regarding it, and aborts the job.

Thus, any process that would have had to disengage before p_f could disengage must disengage or indicate its independence from p_p before p_p can disengage. We therefore conclude that termination cannot be reported prematurely. □

LEMMA 5.3. *In the absence of related-process faults, every surviving process will eventually disengage, and termination will be detected.*

Proof. A failure of a process that is not engaged will not prevent disengagement of any surviving process.

Every engaged failed process p_f has a parent p_p, some (possibly empty) set of other processes that sent it messages, and some (possibly empty) set of other processes to which it sent messages. The parent p_p and all of the other senders will receive failure notifications regarding p_f, allowing them to disregard its debt. All of the non-parents are then free to disengage despite p_f’s failure.

The parent p_p must not disengage until all potential children of p_f are definitively engaged to another parent or have themselves disengaged. By the hypothesis, all of these processes are alive to receive and respond to the parent’s query.

Their responses are as follows:

1. Not engaged: they do not engage as a result of the query, and their reply releases the obligation p_p had to them.
2. Engaged to some process besides p_f: they dispose of their debt to p_f and ignore further messages that may arrive from it, allowing them to eventually disengage from their own parent when appropriate, and their reply releases the obligation of p_p.
3. Engaged to p_f: As above, their past interaction with p_f is written off and will not impede their eventual disengagement. Their reply to p_p indicates that when they disengage, their acknowledgment to p_p will satisfy its obligation to wait for them.

Thus, all recipients of messages from p_f will be able to disengage when appropriate. When those that were engaged to p_f disengage, they will free p_p to disengage as well. □

LEMMA 5.4. *In the presence of a fault involving a process p_{send} and a process p_{recv} to which p_{send} has sent a message, the system will report failure.*

Proof. If p_{send} has sent messages, then it is engaged to some parent p_p, and has sent notifications about all potential p_{recv} to p_p. Thus, after flushing from p_{send}, p_p will have knowledge of the complete set of potential p_{recv}, and will attempt to contact each of them. These contact attempts will either garner an eventual response, indicating that child is not involved in a related-process fault, or an eventual failure notice will reach p_p, at which point it will abort. □

THEOREM 5.5. INDEP *is a correct fault-tolerant termination detection protocol.*

Proof. By lemmas 5.1, 5.2, 5.3 and 5.4, INDEP is correct. □

5.4 Survival Probability

An application using INDEP can survive the concurrent failure of any processes that do not share any communication edges between them. The probability that the protocol can survive the failure of a uniformly random selected set of processes is given by

$$\left[\frac{\binom{n-k}{f}}{\binom{n-1}{f}}\right]^k$$

where n is the number of processes in the job, k is the number of failed processes, and f is the application's fanout [14]. We can combine this formula with failure records from real systems to predict the practical survivability of our protocol. Table 1 shows data from the Cray XT5 system Jaguar at Oak Ridge National Laboratory[1] in combination with the failure probabilities for the protocols presented in this paper with an assumed job size of $n = 1024$. For each algorithm/fanout-parameter combination, we multiply the fraction of faults of a given size by the probability that the termination detector fails to recover from a fault of that size to compute a failure probability from faults of that size, and subtract the sum of those probabilities from 100% to calculate the survivability shown. Other systems for which we have less data available show single-node faults representing 70%–98%, and conservative survivability estimates (taking failures of more than 4 nodes as fatal) ranging from 85% (15% large failures) to 99.98% [14].

The assumption of uniformly distributed faults is an uncertain one. In applications with communication patterns optimized for network locality, this may be problematic. In others settings, such as work stealing, the randomness of the stealing process itself decorrelates the communication graph from system structure.

5.5 Recovery Costs

When an engaged process p_f fails, its parent will (after flushing) have a complete set of the processes to which p_f potentially sent messages prior to its failure. The parent will send a number of recovery messages equal to the size of the set of its potential adoptees, and each of them will send a response message back to the parent. The size of the potential orphan set is bounded by the fanout. Thus, INDEP will send $\mathcal{O}(f)$ control signals in order to recover from each failure.

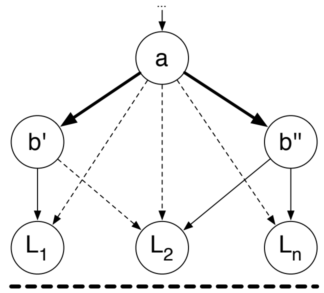


Figure 3: The RELLAZY and RELEAGER protocols handle related process failures providing parent-child interior pairs do not fail simultaneously. In the case depicted, any subset of these processes can fail as long as the failure set does not contain $\{a, b'\}$ or $\{a, b''\}$. $L_1 \ldots L_n$ are exterior processes in the communication graph.

6. Handling Related-Process Failures

We now turn to the problem of faults affecting processes that communicated directly before failing. Given some additional information conveyed in both basic messages and control signals sent during forward execution, the majority of these cases are recoverable (see Figure 3). The non-recoverable case, in which our algorithm reliably reports failure, is one in which it appears to a recovering grandparent that a failed process had a child that was *interior* to the communication graph that also failed.

We define an interior process to mean one that has sent messages to any recipients that have not fully acknowledged them. An exterior process is any process that is not interior. Interior processes must be engaged, by dint of invariant 3, and thus must have a well-defined parent and grandparent. Exterior processes may be engaged or unengaged; if they are engaged, they are leaves of the engagement tree, since they cannot have sent any message that would cause a child to engage to them.

The length of the interval in which the failure of a process and an interior child of that process would be fatal (the vulnerability window) depends on a tradeoff against additional control signals during recovery. In the RELLAZY protocol (§ 6.2), recovery uses the same control signals as in INDEP, but with fewer multi-process fault cases ending in failure, and full recovery is achieved when adopted processes disengage. The RELEAGER protocol (§ 6.3) sends more control signals than INDEP or RELLAZY, but it achieves full recovery once those control signals are processed.

During forward execution, each process process p sends messages to its grandparent p_p indicating its transitions from exterior to interior or vice versa. If p and its parent fail concurrently, p_p flushes those messages from p and checks whether it received an equal number of the two types of messages, or one more interior than exterior, to determine the condition of p when it failed. As we will describe, failed exterior processes can be safely written off, but failed interior processes can not, and are thus fatal.

6.1 Modifications to Fault-Free Execution

In these variants of the protocol, every message carries a note not just of the sender, but also of the sender's parent, so that the recipient knows what process is its grandparent if the sender becomes its parent. This is necessary because each process now transmits additional control signals to its grandparent during forward execution. Those signals convey sufficient information for the grandparent p_p to reconstruct whether the potential grandchild p_c was engaged to a failed child process, and if so, whether p_c was also potentially an interior process.

[1] Personal communication with Terry Jones, ORNL

Nodes Failed	1	2	3	4	5	6	7	8	9	10	11	15	18	26, 86, 126, 338	Survival
Fault %	92.3	3.672	0.942	0.753	0.565	0.094	0.094	0.377	0.094	0.188	0.188	0.282	0.094	$4 \times 0.094 = 0.377$	%
INDEP $f = 2$	0	1.4e-4	1.1e-4	1.8e-4	2.2e-4	5.4e-5	7.4e-5	3.9e-4	1.2e-4	3.1e-4	3.7e-4	9.6e-4	4.3e-4	3.5e-3	99.315
$f = 8$	0	5.7e-4	4.3e-4	6.8e-4	8.2e-4	2.0e-4	2.7e-4	1.3e-3	4.1e-4	9.6e-4	1.1e-3	2.3e-3	8.6e-4	3.8e-3	98.633
$f = 32$	0	2.3e-3	1.6e-3	2.4e-3	2.7e-3	5.8e-4	6.9e-4	3.1e-3	8.5e-4	1.8e-3	1.8e-3	2.8e-3	9.4e-4	3.8e-3	97.466
$f = 512$	0	2.8e-2	9.3e-3	7.5e-3	5.6e-3	9.4e-4	9.4e-4	3.8e-3	9.4e-4	1.9e-3	1.9e-3	2.8e-3	9.4e-4	3.8e-3	93.210
REL*	0	7.2e-5	5.5e-5	8.8e-5	1.1e-4	2.7e-5	3.8e-5	2.0e-4	6.4e-5	1.6e-4	1.9e-4	5.3e-4	2.5e-4	3.3e-3	99.495

Table 1: Distribution of the node counts of concurrent failures on Jaguar, the probability the protocol sees a fault of the given size and fails to recover, and the probability a 1024-process job survives the distribution of concurrent failures using the various algorithms presented.

When a process sends its first message after engaging or after all of its recipients have fully acknowledged it, the process informs its grandparent:

```
def willSendTo(Endpoint successor):
  D++;
  if (D == 1):
    @grandparent { informGrandparentInterior(self, parent) };
    sendFence();
  successors[successor].acksOwed++;
  if (successors[successor].acksOwed == 1):
    @parent { potentialGrandchild(self, successor) };
    requestFailureNotice(successor);
    sendFence();
```

There are two places a process might send `informGrandparentExterior`. The most aggressive is when it receives an acknowledgment that brings its outstanding message count to zero:

```
def gotAck(Endpoint successor, unsigned int count):
  D -= count;
  successors[successor].acksOwed -= count;
  if (successors[successor].acksOwed == 0):
    dropFailureNotice(successor);
    successors.erase(successor);
  if (D == 0):
    @grandparent { informGrandparentExterior(self, parent) };
    sendFence();
  tryDisengage();
```

This creates the shortest intervals in which the failure of both an apparently interior process and its parent would be reported as fatal by the recovering grandparent process. However, it also creates a potentially large number of additional control signals (up to $\mathcal{O}(M)$), depending on how aggressively acknowledgments are transmitted. The other possibility is when it sends an acknowledgment to disengage from its parent, which is also $\mathcal{O}(M)$ in theory, but likely to be lower in practice.

Note that in either case, an exterior signal from process p_c to its grandparent p_g labeled with its parent p_p is always preceded by a similarly labeled and directed interior message, and two interior messages with the same label and destination cannot be sent without a single corresponding exterior signal intervening between them.

At the point of failure of a process p_f, its parent p_p must be able to distinguish whether each of p_f's message recipients p_c was an interior process engaged to p_f, even if p_c fails as well. Using the pairing property between the interior and exterior messages from p_c to p_p, p_p can compute whether p_c was engaged to p_f and interior at the time of its failure by flushing from p_c and then examining the parity between interior and exterior messages. Even parity indicates that every interior message p_c sent was followed by a exterior message, while odd parity indicates that p_c sent one last interior message that was never balanced by a exterior message.

LEMMA 6.1. *In the absence of faults, the related process protocol detects termination correctly.*

Proof. As before, the additional messages and recorded data do not affect detection of termination when no fault occurs. Thus, the correctness of the basic algorithm is maintained. □

6.2 RELLAZY: Lazy Multiprocess Recovery Protocol

The recovery process proceeds as before, but with an increase in the range of failures that a recovering parent p_p can accept from the children of its failed child p_f. In addition to processes that reply to p_p telling of their own non-engagement to p_f, p_p can now write off any process p_c that fails with even parity of interior and exterior messages to p_p after flushing but hasn't responded to the engagement query from p_p. This is safe because it guarantees that p_c was either not engaged to p_f (it sent no interior/exterior messages to p_p) or it was engaged to p_f but had no children of its own (interior and exterior messages from p_c to p_p balanced out). If any p_c fails and flushes out odd parity to p_p, then p_p must report failure because it cannot contract around a failed process for which it was newly responsible.

In this setting, each child p_d of p_c has no knowledge that p_p is its new grandparent, even though p_p successfully adopted p_c as its child and learned about all of p_c's children. Thus, if p_c were to fail in concert with one of its children p_d, p_p would not be able to safely determine whether p_d was an interior process or not, and thus must conservatively report failure. In order to do this, p_p must keep a record that p_c was adopted until p_c disengages. The subsequent failure of such a marked child will be reported as an error. The window of vulnerability in which the system cannot uniformly recover therefore extends from the failure of p_f through the disengagement of all its children p_c from p_p.

LEMMA 6.2. *In the presence of faults,* RELLAZY *does not report termination when it has not occurred.*

Proof. We follow the reasoning of the proof of lemma 5.2, with a change in the handling of failed children of p_f by p_p. When such a child has failed, it is effectively passive for purposes of termination detection. However, its transmissions prior to failure determine whether recovering parents will be allowed to terminate, or report failure. If the interior and exterior messages create odd parity at p_p, then p_p reports failure, and so does not allow early termination. If those message create even parity at p_p we are left with two possible cases:

1. p_c was not engaged to p_f at the time of its failure: if p_c was still engaged at all, it was to some other process that is responsible for its recovery. That process will not disengage and allow termination until its recovery around p_c is complete. This corresponds to case 2 of lemma 5.2.
2. p_c was engaged to p_f at the time of its failure: the even parity indicates that it was not an interior process. Thus, it was not responsible for waiting on any descendants to disengage before it disengaged. The recovering parent p_p has fulfilled its responsibility to delay termination until after p_c disengaged, and can safely disengage without leading to premature termination.

□

LEMMA 6.3. *In the absence of apparent parent/interior-child faults, every surviving process will eventually disengage, and termination will be detected.*

Proof. Consider each process p that receives failure notifications regarding a set of processes. For each of those processes, p has some set of potential grandchildren p_c that it must address. The p_c come in the following varieties:

1. Still alive: they respond to the query from p, and the eventual disengagement of p and these grandchildren is as described by lemma 5.3.
2. Dead, with the resulting flush of messages providing
   (a) Even parity: p_c was not engaged to p's child at the time of failure, or was so engaged but had no potential children and so could not be interior. p can safely write them off, as it has no obligation to await disengagement of any children they may have had.
   (b) Odd parity: the hypothesis that the fault did not involve an apparent parent/interior-child pair was violated, and p will report an error.

In all cases satisfying the condition, p and all recipients of messages from failed processes are therefore allowed to disengage when their other obligations are satisfied, and termination will be reported. □

LEMMA 6.4. *In the presence of a fault involving a process p_{send} and an interior process p_{recv} which was engaged to p_{send}, the system will report failure.*

Proof. The parent p_p of p_{send} will receive a notification that p_{send} failed with a flush of messages. In the recovery process, p_p will request failure notification from all processes to which p_{send} sent messages, including p_{recv}. By the construction of the message sending code and the pairing property of the interior and exterior messages, p_{recv} must have transmitted odd parity of these messages to p_p. Upon flushing, p_p will observe this odd parity, conclude that the fault included a parent/interior-child pair, and report failure. □

THEOREM 6.5. RELLAZY *is a correct fault-tolerant termination detection protocol.*

Proof. By lemmas 6.1, 6.2, 6.3 and 6.4, RELLAZY is correct. □

6.2.1 Recovery Costs

The recovery communication resulting from a failure in RELLAZY is the same as in INDEP. Thus, RELLAZY sends at most $\mathcal{O}(f)$ control signals in order to recover from each failure.

6.3 RELEAGER: Eager Multiprocess Recovery Protocol

We now address recovery from failures of the following form:

- a process p_f fails, causing its parent p_p to initiate recovery,
- a process p_c was a child of p_f, and is adopted by p_p,
- p_c sent one or more messages to an exterior process p_d before the failure of p_f and subsequent recovery steps, where
- p_c and p_d subsequently fail concurrently.

As noted earlier, the challenge in this situation is that p_p will never have been sent the information necessary to determine whether its adopted potential grandchild p_d is an interior child of p_c or not. To correct that deficiency, the recovery protocol can exchange additional signals that convey the necessary information.

This eager recovery protocol variant still begins the same way, with processes that sent messages to a failed process p_f being informed of the failure and flushing from it, and its parent p_p querying each of its potential grandchildren p_c. We depart from the lazy recovery protocol in that the responses from p_c will no longer be returned immediately. Instead, they send a message to each of their own potential children p_d indicating their new potential grandparent p_p.

Each of those p_d respond to p_c indicating whether they are engaged to p_c, and if so, whether they are interior. The p_d that are engaged to p_c update their grandparent record to point to p_p instead of the now-dead p_f. When p_c receives all of those responses, it sends an aggregated response to p_p with the list of actually engaged children and an indicator of which of those children are interior.

When p_p receives the response message from p_c, the list of grandchildren is recorded as before, but with the interior bits used to initialize its view of each grandchild's interior/exterior parity. Essentially, this reconstructs the unpaired interior messages that those p_d would have sent to p_f before its failure. When p_p receives responses from all of its adopted children p_c, the system is no more susceptible to failure than it was prior to the failure of p_f– the window of vulnerability is closed.

Note that this recovery protocol races with potential disengagement of the affected processes. This race is not only tolerated, but actually aids resilience. Any process that disengages before the protocol finishes recovery is one fewer process whose failure will bring down the entire job.

THEOREM 6.6. RELEAGER *is a correct fault-tolerant termination detection protocol.*

Proof. The reasoning of lemmas 6.1, 6.2, 6.3 and 6.4 applies to RELEAGER as well, and thus RELEAGER is correct. □

6.3.1 Recovery Costs

Each failed engaged process will cause its parent to send a message to each of the failed processes' potential children. Failures among those children do not increase the message count. Each of the actual grandchildren sends a message to their own potential children (i.e. the potential great-grandchildren), and gets a response from each. The grandchildren then each send a response to the grandparent. Each of the grandparent and grandchildren may have $\mathcal{O}(f)$ messages to send, and there may be up to f grandchildren, giving a total message count bound of $\mathcal{O}(f^2)$. These messages are distributed such that each process handles at most $\mathcal{O}(f)$ of them.

6.4 Survival Probability

Out of an N-process job, consider a failure of k processes. By lemma 6.3, the algorithm can survive the failure of any one process, as long as its parent does not also fail with it. Out of the failed subset, consider the probability that a single process v has failed along with its parent. The number of k-process failure sets containing v is

$$\binom{N-1}{k-1}$$

The number of these that also contain v's parent is

$$\binom{N-2}{k-2}$$

Thus, the probability that the failure set will contain v's parent, given that it contains v, is given by their ratio

$$\frac{k-1}{N-1}$$

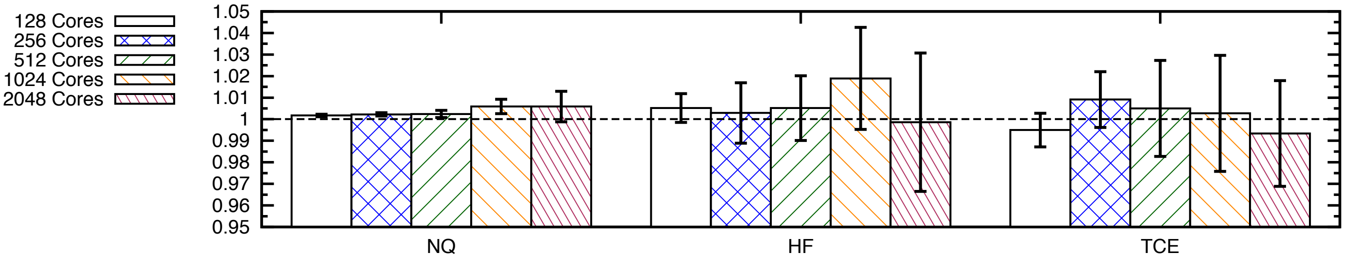
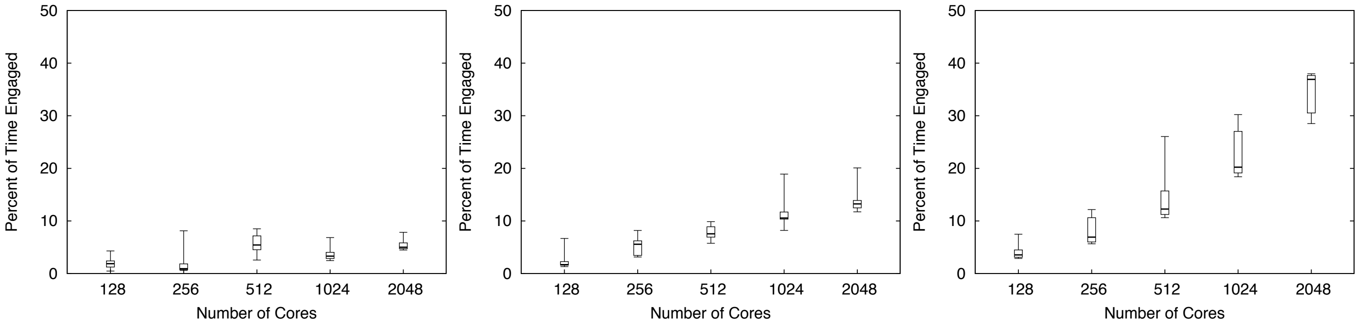


Figure 4: The overhead of our related-process fault tolerance protocols (RelLazy or RelEager). Each bar shows the ratio of the average execution time using the protocol vs. average execution time without. Sample size is 24 runs of each application, at each scale, both with and without our protocol enabled. The whiskers represent the error in the difference of means at 99% confidence, using a Student's t-test.



(a) NQ

(b) HF

(c) TCE

Figure 5: For three benchmarks, the percent of total execution time (in terms of processor-seconds) spent engaged: the time when a process is interior in the communication graph. For the RelLazy and RelEager schemes, a process and its parent cannot fail simultaneously during this time or the protocol will not survive the fault. The vertical lines stretch between the minimum and maximum values; the box spans between the 25th and 75th percentile; the horizontal line spanning the box indicates the median.

The probability that the system survives this process's failure is then $1 - (k-1)/(N-1)$. Note that this directly reproduces the qualitative result that the algorithm always survives single-process failures (i.e. $k = 1$ fails with probability 0).

Given that k processes failed, and pessimistically assuming that none of them shares a parent (i.e. maximizing the cases that would lead to failure), the probability that the entire system survives is

$$\left(1 - \frac{k-1}{N-1}\right)^k$$

As with Indep, we can combine these probability estimates with real failure data to determine how resilient the algorithms will be in practice. The results are shown in Table 1. Unlike Indep, RelLazy and RelEager do not become more vulnerable as fanout increases, because each process may have many communication partners, but has at most one parent. As the table shows, the two related-process protocols are more resilient than Indep even in the most lenient fanout-2 case.

7. Experimental Results

7.1 Experimental Setup

We performed our experiments on Intrepid, a 40960-node IBM Blue Gene/P. Each node consists of a four-core 850MHz PowerPC 450 processor and 2 GB of DDR2 memory. We compiled our code with the IBM XL C/C++ Advanced Edition for Blue Gene/P, V9.0.

Our codes use an active-message-based runtime [17] implemented using MPI primitives with distributed-memory work stealing to balance work between cores during an iteration. We run one process per SMP node in a multi-threaded configuration, with one thread per core. One core is used as a server that polls the network and executes active messages. The other three cores maintain a local deque of tasks that is optimized for efficient remote steal attempts and local task execution [4, 11].

7.2 Benchmarks

We evaluate our fault tolerance scheme with three benchmarks that involve parallel task collections. For the N-Queens benchmark (NQ), each task at depth d recursively tries to place the next queen at depth $d + 1$ creating up to N new tasks. At a certain depth s, a task stops producing more tasks and executes sequentially. The Hartree-Fock (HF) method is an algorithm from computational chemistry that forms the basis for several higher-level theories such as Coupled Cluster theory and Møller-Plesset perturbation theory. Our benchmark is comprised of the two-electron contribution component of Hartree-Fock. The matrices used for the computation (`schwarz`, `fock`, and `dens`) are distributed using a global address space and each task reads/writes the necessary portions. Tensor Contraction Expressions (TCE) constitute the entirety of Coupled Cluster methods, used in accurate descriptions of many-body systems in diverse domains. We benchmark a tensor contraction where the matrices are also distributed in a global address space.

7.3 Fault-free Execution Overhead

Our fault-tolerance protocols incur extra message sends during fault-free execution. All three protocols (Indep, RelLazy, RelEager) require a given process to send a message to its parent informing it of a potential grandchild before it actually sends a message to a new child. The related-process failure protocols RelLazy and RelEager require a process to send a message to its grandparent before it possibly engages a child by sending it a message. We measure and plot the overhead of the related-failure case in Figure 4. The slightly less overhead induced in the Indep protocol is statistically indistinguishable from the related-process case so we omit it. Figure 4 displays the ratio of execution time using the fault tolerance protocol versus normal execution without any fault tolerance.

For the NQ benchmark, the overhead is under 0.5% and is mostly within the error. The NQ benchmark is very well-behaved since the tasks are fairly uniform and do not access global data. The HF and TCE benchmarks have higher run-to-run variation, and the overhead we induce with our protocol is clearly within the error.

7.4 Measuring Exterior Processes

The RELLAZY and RELEAGER protocols can recover from a fault when the processes that fail do not include a parent/interior-child pair. Whether a process is an interior child is transient over time: processes may switch between exterior and interior over the course of a run depending on the communication patterns of the application. For the three benchmarks that are presented, we measure the amount of time that processes are in the exterior state compared to the total execution time of the application. The communication of these applications is primarily driven by the random work stealing employed, but varies depending on the duration of tasks and parallel slack available in different regions of the computation.

Figure 5 displays the percent of total execution time spent engaged after sending a message, which is an upper-bound on time spent as an interior process. Recall that a failed process can only be part of an unhandled related-process fault while it is interior. We show for all three benchmarks that this percentage of time is under 40 percent, but it is application dependent in terms of scaling behavior.

8. Conclusion

We have described a new approach to producing a fault-tolerant termination detection algorithm based on the original parental-responsibility algorithm described by Dijkstra and Scholten [3]. The three algorithms we describe have worst-case overhead and recovery costs measured in terms of the application's *process fanout*, which is upper-bounded by the DS algorithm's provably-optimal $\mathcal{O}(M)$ bound and by the number of processes in the system. In practice, especially in scalable HPC applications, it is often much smaller. Thus, the overhead for making termination detection fault tolerant is lower than the overhead of termination detection itself. Additionally, all overhead costs are distributed in a localized, scalable manner, avoiding the centralization that makes other fault-tolerant termination detectors unsuitable for HPC.

Through benchmark results, we have shown that the practical overhead costs are minimal. We use real system fault data to show that our algorithms are highly survivable in the face of the faults they are likely to encounter. Therefore, we conclude that our algorithms are well-suited to large-scale parallel applications.

Acknowledgements

The authors were supported by DOE grant DE-SC0006706 and NSF grants ITR-HECURA-0833188 and OCI-0725070. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2009. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504215.

[2] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410 – 416, 2009. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.12.002.

[3] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1–4, 1980.

[4] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11. ACM, 2009. doi: 10.1145/1654059.1654113.

[5] F. Freiling, M. Majuntke, and N. Mittal. On detecting termination in the crash-recovery model. In A.-M. Kermarrec, L. Boug, and T. Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 629–638. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74465-8.

[6] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors, 2002.

[7] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 159–168. ACM, 2010. doi: 10.1145/1693453.1693476.

[8] L. Kalé and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[9] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[10] T.-H. Lai and L.-F. Wu. An $(N-1)$-resilient algorithm for distributed termination detection. *Parallel and Distributed Systems, IEEE Transactions on*, 6(1):63–78, Jan 1995. doi: 10.1109/71.363410.

[11] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 137–148. ACM, 2012. doi: 10.1145/2287076.2287103.

[12] W. Ma and S. Krishnamoorthy. Data-driven fault tolerance for work stealing computations. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 79–90. ACM, 2012. doi: 10.1145/2304576.2304589.

[13] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987. doi: 10.1007/BF01782776.

[14] E. Meneses, X. Ni, and L. V. Kale. A Message-Logging Protocol for Multicore Systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[15] N. Mittal, F. C. Freiling, S. Venkatesan, and 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*, DISC'05, pages 93–107, 2005.

[16] S. Venkatesan. Reliable protocols for distributed termination detection. *Reliability, IEEE Transactions on*, 38(1):103–110, Apr 1989. ISSN 0018-9529. doi: 10.1109/24.24583.

[17] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[18] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: a generalized active message framework. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, 2010. doi: 10.1145/1854273.1854323.

[19] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244. ACM, 2011. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995934.