


# Resilient Optimistic Termination Detection for the Async-Finish Model

**Sara S. Hamouda**

Australian National University, Canberra, ACT 2601, Australia  
sara.salem@anu.edu.au

**Josh Milthorpe**

Australian National University, Canberra, ACT 2601, Australia  
josh.milthorpe@anu.edu.au  
 0000-0002-3588-9896

## Abstract

[Paper submitted to DISC 2018 on 12 May 2018 - currently under review]

Driven by the consistent trend of increasing core count and decreasing mean-time-to-failure in supercomputers, HPC runtime systems must improve support for dynamic task-parallel execution and resilience to failures. The async-finish task model, adapted for distributed systems as the asynchronous partitioned global address space programming model, provides a simple way to decompose a computation into nested task groups, each managed by a 'finish' that signals the termination of all tasks within the group. A failure impacting one finish scope does not propagate to other finish scopes by default, providing an opportunity for local recovery, an important feature for resilient large-scale computations.

Prior work for resilient finish termination demonstrated that simple mechanisms for recovering the control structure of a computation can be achieved by adding more termination signals over the optimal number of signals needed in a resilient failure-free execution. In this paper, we propose *optimistic finish*, the first message-optimal resilient termination detection protocol for the async-finish model. By avoiding the communication of certain task and finish events, this protocol allows uncertainty about the global structure of the computation which can be resolved correctly at failure time, thereby reducing the overhead for failure-free execution.

We implemented two variants of optimistic finish, a centralized and a distributed implementation, and compared them to corresponding pessimistic resilient finish implementations in the X10 language. Performance results using micro-benchmarks, and the LULESH hydrodynamics proxy application show significant reductions in resilience overhead with the optimistic finish compared to the pessimistic finish. Our proposed optimistic finish protocol is applicable to all task-based runtime systems offering automatic termination detection for arbitrary task graphs.

**2012 ACM Subject Classification** Concurrent Programming → Distributed programming, Reliability → Fault-tolerance

**Keywords and phrases** Fault tolerance, termination detection, APGAS, async-finish, X10

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.

**Acknowledgements** We would like to thank David Grove and Olivier Tardieu of IBM T. J. Watson Research Center for useful discussions which clarified our understanding of the semantics of resilient X10 and the underlying protocol, and guided our design of optimistic finish. The research was undertaken with the assistance of resources from the National Computational Infrastructure (NCI) and NECTAR research cloud, which are supported by the Australian Government. We are deeply grateful to Peter Strazdins for facilitating our access to NCI's resources and for reviewing this paper.



© Sara S. Hamouda and Josh Milthorpe;  
licensed under Creative Commons License CC-BY  
42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. ; pp. 1–16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

Recent advances in high-performance computing (HPC) systems have resulted in greatly increased parallelism, with both larger numbers of nodes, and larger numbers of computing cores within each node. With this increased system size and complexity comes an increase in the expected rate of failures. Programmers of HPC systems must therefore address the twin challenges of efficiently exploiting available parallelism, while ensuring resilience to component failures.

Dynamic task-parallel execution models present an attractive approach to both of these challenges. A dynamic computation evolves by generating asynchronous tasks that form an arbitrary directed acyclic graph. A key feature of such computations is termination detection (TD): determining when all tasks in a subgraph are complete. In an unreliable system, additional work is required to ensure that termination detection is correct even in the presence of component failures. Task-based models for use in HPC must therefore support resilience through efficient fault-tolerant termination detection protocols.

Fault-tolerant termination detection has been widely studied for diffusing computations involving a single termination scope [3, 5, 6]. Most of the algorithms assign a parental responsibility on the intermediate tasks, requiring each intermediate task to signal its termination only after its successor tasks terminate. The root task detects global termination when it receives the termination signals from its direct children. This execution model is very similar to Cilk’s spawn-sync model, where a task calling ‘sync’ waits for join signals from the tasks it directly spawned. While having the intermediate tasks as implicit TD coordinators is favorable for balancing the traffic of TD among tasks, it adds unnecessary blocking at each task that is not needed for correctness.

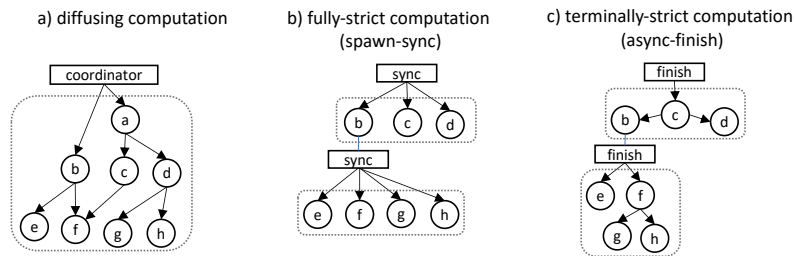
The *async-finish* task model, implemented by APGAS languages such as X10 and HabaneroC, provides a more general model for expressing dynamic computations. It supports multiple termination scopes that can be nested or side-by-side, and avoids blocking intermediate tasks for the sake of termination detection. Previous research related to the *async-finish* programming model has mostly focused on productivity or performance, and ignored issues related to resilience. However, a resilient TD protocol was recently developed for the X10 language [2]. This protocol maintains a consistent view of the task graph across multiple processes, which can be used to reconstruct control flow in case of a process failure. It adds more termination signals over the optimal number of signals needed in a resilient failure-free execution for the advantage of simplified failure recovery. Since it favors failure recovery over normal execution, we describe this protocol as ‘pessimistic’.

In this work, we review the published protocol for X10, and demonstrate that the requirement for a consistent view results in a high performance overhead for failure-free execution. We propose the ‘optimistic finish’ protocol, an alternative message-optimal protocol that relaxes the consistency requirement, resulting in faster failure-free execution with a moderate increase in recovery cost.

A complete implementation of the protocol is publicly available in a GitHub repository [11]. We also developed a formal specification of the proposed protocol using TLA+ and made it publicly available [12]. Using the TLA+ model checker, we verified the correctness of the protocol over a 3-level task tree with a branching factor of 2, where a failure can impact one place at any point in the execution. This abstract task tree captures all the interesting communication patterns that impacts *async-finish* termination detection.

We begin with a review of dynamic task-based computation models and approaches to resilient termination detection. Next, Section 5 presents an overview of the X10 programming model; Section 6 describes Resilient X10 including the pessimistic protocol used in previous implementations of Resilient Finish; Section 7 presents our proposed optimistic protocol; and Section 8 evaluates the performance of the new protocol on a set of micro-benchmarks and two more realistic application benchmarks.

## 2 Background: Dynamic Computation Models



**Figure 1** Dynamic computation models that require termination detection. A dotted-box represents a single termination scope. Circles represent work units, which are processes in (a), and tasks in (b) and (c).

In the context of termination detection, dynamic computations can be classified into three main models. The standard model of termination detection is the diffusing computation model [3](Figure 1-a). In that model, the computation starts by activating a coordinator process that is responsible for signaling termination. Other processes are initially idle and can only be activated by receiving a message from an active process. An active process can become idle at any time. The computation terminates when all processes are idle, and no messages in transit to activate an idle process [13]. A diffusing computation has a single termination scope that is signaled by the coordinator process.

Computations that entail nested termination scopes are generally classified as fully-strict or terminally-strict [1] (Figure 1-b,c). Blumofe and Leiserson [1] describe a fully-strict task DAG as one that has fork edges from a task to its children, and join edges from each child to its direct parent. In other words, a task can only wait for other tasks it directly forked. On the other hand, a terminally-strict task DAG allows a join edge to connect a child to any of its ancestor tasks, including its direct parent, which means a task can wait for other tasks it directly or transitively created. Cilk's spawn-sync programming model, and X10's async-finish programming model are the most prominent representations of the fully-strict and terminally-strict computations, respectively. By relaxing the requirement to have each task to wait for its direct successors, the async-finish model avoids unnecessary synchronization while creating dynamic irregular task trees, that would otherwise be imposed by spawn-sync. Guo et al. [4] provide more details on X10's implementation of the async-finish model, contrasting it to Cilk's spawn-sync model.

From the diagrams in Figure 1, we can identify the following relations between the three models: 1) a diffusing computation is an async-finish computation with a single finish scope, and 2) a spawn-sync computation is an async-finish computation where each finish is governing direct tasks only. Thus async-finish is a super-set of both diffusing and spawn-sync computations, and in theory async-finish TD protocols are directly applicable to the other two models. Although a single finish scope can be considered as a diffusing computation, most diffusing TD protocols force parent/child synchronization constraints between intermediate tasks that make them identical to the spawn-sync model, and therefore not always applicable for async-finish computations.

When failures occur, nodes in the computation tree are lost, resulting in sub-trees of the failed nodes breaking off the computation structure. Fault-tolerant termination detection protocols aim to reattach those sub-trees to the remaining computation to facilitate termination detection.

## 3 Related Work

Termination detection of a dynamic computation is a well-studied problem in distributed computing, having multiple protocols proposed since the 1980s. Interested readers can refer to [7] for a comprehensive survey of many TD algorithms.

The DS protocol, presented by Dijkstra and Scholten [3], was one of the earliest TD protocols for diffusing computations and has been extended in different ways for adding resilience. It is a *message-optimal* protocol such that for a computation that sends  $M$  basic messages, DS adds exactly  $M$  control messages to detect termination. The control messages are acknowledgements that a process must send for each message it receives. DS enforces the following constraints to ensure correct termination detection. First it requires each process to take as its parent the origin of the first message it received while being idle. Second a child process must delay acknowledging the parent messages until after it has acknowledged all other predecessors and received acknowledgements from its successors. By having each parent to serve as an implicit coordinator for its children, DS ensures that termination signals flow correctly from the leaves to the top of the tree. That also transforms the diffusing computation to a spawn-sync computation. Fault-tolerant extensions of the DS algorithm are presented in [5, 6].

Lai and Wu [5] describe a resilient protocol that can tolerate the failure of almost the entire system without adding any overhead for failure-free execution. The idea is that each process (idle and active) detecting a failure must detach from its parent, adopt the coordinator as its new parent, and share its local failure knowledge with its parent and the coordinator. On detecting a failure, the coordinator expects all processes to send termination signals directly to it. The protocol introduces a sequential bottleneck at the coordinator process when failures occur, which limits its applicability to large-scale HPC applications.

Venkatesan [13] presented a TD algorithm that relies on replicating the local termination state of each process on  $k$  leaders to tolerate  $k$ -process failures. The protocol assumes that the processes are connected via first-in-first-out channels and that a process can send  $k$  messages atomically to guarantee replication consistency. Unfortunately these features are not widely available in large-scale HPC systems, which limits the applicability of this algorithm.

Liffander et al. [6] took a practical approach for resilient TD of a diffusing computation. Based on the assumption that multi-node failures are rare in practice, and that the probability of  $k$ -node failure decreases as  $k$  increases [8, 10], they designed three variants of the DS protocol that can tolerate most but not all failures. The INDEP protocol tolerates the failure of a single process, or multiple unrelated processes. It requires each parent to have a consistent view of its successors and their successors. To achieve this goal, each process notifies its parent of its potential successor before sending a message to it. Two more protocols were proposed to address related-process failures. In addition to the notifications required in INDEP, these protocols also require each process to notify its grandparent when its state changes from interior (non-leaf node) to exterior (leaf node) or vice versa. A failure that affects both an interior node and its parent is *fatal* in these protocols. Two special network services are required for correct implementation of the three protocols: a network send fence and a fail-flush service.

To the best of our knowledge, the only prior work addressing resilient termination detection for the async-finish model is that of Cunningham et al. [2]. They describe a protocol which separates the concerns of termination detection and survivability. They use a resilient store to maintain critical TD data. As the computation evolves, signals are sent to the resilient store to keep it as updated as possible with potential changes in the control structure. The used signals enable the resilient store to independently recover the control structure when failures occur, however, they impose high performance overhead in failure-free execution. Unlike previous work, this protocol does not require any special network services other than failure detection.

Our work combines features from [6] and [2] to provide a practical low-overhead termination detection protocol for the async-finish model. Assuming multi-node failures are rare events, we designed a message-optimal ‘optimistic’ protocol that significantly reduces the resilience overhead in failure-free scenarios.

## 4 Message-Optimal Async-Finish Termination Detection

In this section, we aim to identify the optimal number of control messages required for correct async-finish termination detection in both non-resilient and resilient implementations.

We assume a distributed finish block that includes nested async statements that create tasks at different locations. Messages sent to fork these tasks at their intended locations are referred to as *basic messages*. For example, in the task graphs in Figures 2 and 3, three basic messages are sent to fork tasks a, b, and c. The additional messages used by the system for the purpose of termination detection are referred to as *control messages* (shown as dotted lines in the figures). We consider the basic messages as the baseline for any termination detection protocol, thus an optimal protocol will add the minimum number of control messages as overhead. In resilient mode, we build on Cunningham et al.'s design in which the finish objects are held in a resilient store as shown in Figure 3.

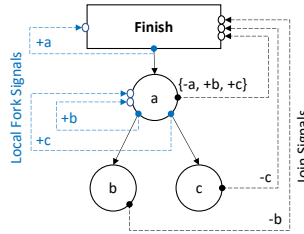


Figure 2 Message-optimal non-resilient finish

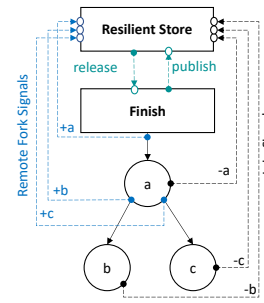


Figure 3 Message-optimal resilient finish

As a finish scope evolves by existing tasks forking new tasks, finish needs to update its knowledge of the total number of pending tasks so that it can wait for their termination. We refer to this number as the *global count* or *gc* in this paper. The global count is incremented when a new task is forked, and decremented when a running task joins. When the last task joins, the global count reaches zero and finish is released. We use the terms *fork signal* and *join signal* to refer to the control signals created just before forking a new task and just after a task terminates, respectively. Multiple signals from the same source may be packed in one message for better performance.

► **Lemma 1.** A message-optimal **non-resilient** finish adds at most one TD control message per task.

**Proof.** Finish detects termination only after all tasks join. Thus sending a join signal per task is unavoidable for correct termination detection. To avoid sending messages other than those carrying the join signals, a task can save the fork signals of its successor tasks locally and send them with its join signal in the same message. Assuming that failures do not occur, each task must eventually join. Thus finish will eventually receive a fork signal and a join signal for each task, guaranteeing correct termination detection. ◀

Figure 2 demonstrates this message-optimal approach for termination detection. When task a forks tasks b and c, it delays sending their fork signals (+b, and +c) until it joins. At this point, it packs its join signal (-a) with the delayed fork signals and sends one message containing the three signals (-a, +b, +c). Note that delaying the fork signals may result in tasks joining before their fork signals are received by finish. A correct implementation must delay termination until each fork is matched by a join and each join is matched by a fork.

The worst-case scenario occurs when each task spawns at a different location. Fewer control messages are required if multiple tasks spawn at the same location. For example, a finish sending

1000 tasks to one remote place can receive one large message notifying the termination of all 1000 tasks, rather than receiving 1000 individual messages.

► **Lemma 2.** *A message-optimal **resilient** finish adds at most two TD control messages per task and two control messages per finish in failure-free execution.*

**Proof.** As in the proof of Lemma 1, sending one join signal per task is unavoidable for correct termination detection. In the presence of failures, tasks may be lost at arbitrary times leaving behind active successor tasks. Lazy notification of fork signals (as in the non-resilient finish above) would lead finish to lose track of a successor task if its predecessor failed before sending the successor's fork signal. Therefore, fork signals must be sent eagerly prior to sending remote tasks. As a result, a maximum of two messages are needed per task for correct termination detection in the presence of failure – a fork message and a join message. Since the finish itself may be lost due to a failure, its state needs to be saved in a resilient store that can survive failures. This requires sending a message to the resilient store to notify the creation of a new finish scope. A corresponding message is needed from the resilient store to the finish when termination is finally reached. Thus, a maximum of two messages are needed per finish in failure-free execution. ◀

Figure 3 demonstrates the above message-optimal approach for resilient termination detection. Note that in counting the messages, we do not take into account the messages that the resilient store may generate internally to guarantee reliable storage of data. While a centralized resilient store may not introduce any additional communication, a replication-based store will introduce communication to ensure data consistency between replicas.

## 5 X10: An overview

We used X10 as a basis for our study of termination detection protocols for the async-finish model. In this section, we give an overview of X10's programming model, the runtime APIs and mechanisms used for tracking dynamic tasks, and finally describe X10's non-resilient finish protocol.

### 5.1 Programming Model

X10 is an asynchronous partitioned global address space language. It models a parallel computation as a global address space partitioned among places. Each place is a multi-threaded process with threads cooperating to execute tasks spawned locally or received from other places. Initially, all places are idle except place zero. It starts by spawning a task that executes the main method within a finish scope. This finish is the top of the task DAG, and is used for signaling final termination.

An X10 program dynamically generates an arbitrary task DAG by nesting **async**, **at**, and **finish** constructs. The **async** construct spawns a new task at the current place such that the spawning task (the predecessor) and the new task (the successor) can run in parallel. To spawn an asynchronous task at a remote place  $p$ , **at** is used with **async** as follows: (**at** ( $p$ ) **async**  $S$ );. An error raised while spawning or executing  $S$  is thrown in an Exception object. The **finish** block is used for synchronization; it defines a scope of coherent tasks and waits for their termination. Exceptions thrown from any of the tasks are collected at the finish, and are wrapped in a MultipleExceptions object after finish terminates. Note that **at** is a synchronous construct, which means that **at** ( $p$ )  $S$ ; (without **async**) does not return until  $S$  completes at place  $p$ . Unlike **finish**, **at** does not wait for successor asyncs spawned by  $S$ . A place may hold references to objects hosted at remote places using the `x10.lang.GlobalRef` type. To access a remote object using its global ref  $gr$ , a task must shift to the object's home as follows: (**at**( $gr$ )  $gr().doSomething()$ );. This model enables moving computations to data in X10 programs.

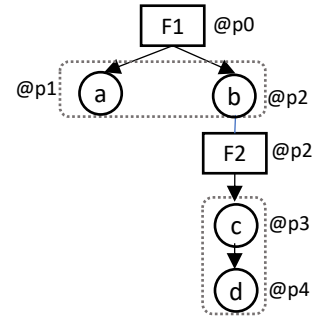


Figure 4 shows a sample X10 program, with the corresponding task DAG. The statements at lines 9 and 12 are added to highlight the happens-before relations that finish imposes between statements at different places. These relations must always be maintained, even in the presence of failures.

```

1 finish { /* F1 */
2   at (p1) async { /* a */ }
3   at (p2) async { /* b */
4     finish { /* F2 */
5       at (p3) async { /* c */
6         at (p4) async { /* d */ }
7       }
8     }
9   printf('tasks c and d completed');
10 }
11 }
12 printf('tasks a, b, c, and d completed');

```



■ **Figure 4** A sample X10 program and the corresponding task graph.

## 5.2 Distributed Task Tracking

Termination detection requires tracking the creation and termination of tasks. X10 defines a simple programming interface for tracking the tasks of each finish block. In the following, we describe the essence of the programming interface using pseudocode.

A termination detection protocol is defined by providing concrete implementations of the abstract classes `Finish` and `LocalFinish` shown in Listing 1. One instance of `Finish` with a globally unique id is created for each finish block to maintain a global view of the distributed task graph. It includes a latch that is used to block the task that started a finish block until all the tasks within that block terminate. The function `wait` will be called at the beginning of the finish block. When all tasks terminate, the function `release` will be called to release the blocked task. The runtime system links the finish objects in a tree structure by providing to each finish object a reference to its parent finish.

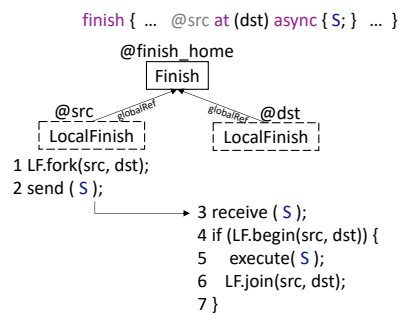
Each visited place within a finish block will create an instance of type `LocalFinish` to track task activities done locally. It holds a global reference to the global `Finish` object to notify it when changes in the task graph occur to enable `Finish` to have an up-to-date view of the global control

■ **Listing 1** X10 Finish API

```

1 abstract class Finish(id) {
2   val latch:Latch;
3   val parent:Finish;
4   def wait() { latch.wait(); }
5   def release() { latch.release(); }
6 }
7 abstract class LocalFinish(id) {
8   val gr:GlobalRef[Finish];
9   def fork(src, dst):void;
10  def begin(src, dst):bool;
11  def join(src, dst):void;
12 }

```



■ **Figure 5** Updating finish while invoking remote tasks

structure. The abstract class `LocalFinish` defines three interfaces to track task events: `fork`, `begin`, and `join`<sup>1</sup>. Figure 5 shows the invocation of the three task events when a source place `src` spawns a task at a destination place `dst`. On spawning a new task, the source place calls `fork` to notify finish of a potential new task, then it sends the task to the destination place. On receiving a task, the destination place calls `begin` to determine whether or not the task is valid for execution. If the task is valid, the destination place executes it, then calls `join` to notify task termination. If the task is invalid, the destination place drops it. In failure-free execution, `begin` may assume that all incoming tasks are valid. However, when failures are expected, `begin` may need to consult its governing `Finish` object to determine the validity of the received task. We will clarify that as we describe the finish protocols in the following sections.

In this paper, we describe each protocol in terms of the variables of the `Finish` and `LocalFinish` objects, and the implementations of the three task events `fork`, `begin`, and `join`. In the included pseudocode, we use the notation `@F[id]`, `@LF[id]`, and `@RF[id]` to refer to accessing a remote `Finish` object, `LocalFinish` object and `ResilientFinish` object, respectively.

■ Listing 2 Non-resilient Finish

```

1 class NR_Finish(id) : Finish {
2   gc:int=0; // global count
3   live:int[places]={0};
4   def merge(remoteLive) {
5     for (p in places) {
6       live[p] += remoteLive[p];
7       gc += remoteLive[p];
8     }
9     if (gc == 0) release();
10  }
11 };

```

■ Listing 3 Non-resilient Local Finish

```

12 class NR_LocalFinish(id) :
13   LocalFinish {
14     lc:int=0; //local count
15     live:int[places]={0};
16     def fork(src, dst) {
17       live[dst]++;
18     }
19     def begin(src, dst) {
20       lc++; return true;
21     }
22     def join(src, dst) {
23       live[d]--; lc--;
24       if (lc == 0)
25         @F[id].merge(live);
26     }
27 };

```

### 5.3 Non-Resilient Finish Protocol

The default TD protocol in X10 is non-resilient. It assumes that the finish and local finish objects are available for the lifetime of the finish block, and that each spawned task will eventually join. Ignoring the garbage collection messages that the runtime uses to delete the `LocalFinish` objects at the end of a finish block, the non-resilient finish protocol is message-optimal. It applies the reporting mechanism described in Figure 2 to use a maximum of one TD message per task. The pseudocodes in Listing 2 and Listing 3 capture the details of the protocol implementation. The finish object maintains a global count, `gc`, and an array `live` stating the number of live tasks at each place. The local finish (LF) object maintains a local count `lc` stating the number of tasks that began locally, and a local live array that states the number of spawned tasks to other places. The `begin` event, called at the receiving place, increments the local count without consulting the finish object, because this protocol is not prepared for receiving invalid tasks due to failures. When all the local tasks terminate (i.e. when `lc` reaches zero), LF passes its local `live` array to the finish object. Finish merges the passed array with

<sup>1</sup> We modified X10's `LocalFinish.join()` interface changing it from `join(dst)` to `join(src, dst)` to implement the optimistic finish protocol



its own live array and updates gc to reflect the current number of tasks. Termination is signalled when gc reaches zero.

## 6 Resilient X10

Resilient X10 [2] is a recent extension to X10 adding support for user-level fault tolerance. It focuses on fail-stop process failures, in which a failure of a place results in immediate loss of its tasks and data and prevents it from communicating with other places. When such failures occur, tasks and finish objects may be lost resulting in disconnecting parts of the computation from the task DAG. A key contribution of the resilient X10 work is the design of a termination detection protocol that is capable of repairing the computation DAG after failures. It applies an adoption mechanism that enables a grandparent **finish** to adopt its orphaned grandchildren tasks to attach them back to the computation's DAG.

### 6.1 Challenges of Async-Finish Termination Detection Under Failure

Distributed task graphs with nested termination scopes impose the following challenges on termination detection in the presence of failures:

1. Loss of termination detection metadata: as the computation evolves, termination detection metadata such as **Finish** and **LocalFinish** objects are created at different places. The failure of a place may result in losing some of these objects that are critical for correctly signaling termination.
2. Inconsistent failure knowledge: when a failure occurs, the places may detect the same failure at different times leading them to take actions that can violate the consistency of the system. For example, a finish that is aware of a place failure may decide to exclude the tasks sent by that place from its global count, while another place that is unaware of the failure may decide to execute the tasks that finish has already excluded.
3. Uncertainty about orphaned tasks: failures may result in orphaned tasks if the finish that responsible for tracking their activities is lost. Accurate identification of these orphan tasks is necessary for recovering the task DAG.
4. Uncertainty about tasks spawned from a dead source: if a source place died after it had notified **finish** of its intention to send a remote task, this task may be in one of three states: a) not transmitted because the source place died before sending it, b) fully-transmitted and will be received by the destination, or c) partially-transmitted and will be dropped at the destination due to message corruption.

To address the first two challenges, Cunningham et al. [2] proposed using a *resilient store* that can save the data reliably and live beyond failures. The design of the resilient store is orthogonal to the termination detection protocol, thus different stores (i.e. centralized/distributed, disk-based/memory-based, native/out-of-process) can be used. However, the survivability of the protocol implementation is limited by the survivability of the store. To avoid losing TD metadata, a **ResilientFinish** object will be created for each finish block to track the state of its tasks. Rather than communicating with the **Finish** object that may be lost at any time, task events are communicated to the resilient finish object. For the second challenge, **ResilientFinish** can serve as the reference for identifying the status of places. Places can agree on the failure status of places by querying of the **ResilientFinish** object, rather than relying on their own local knowledge.

Delegating the responsibility of termination detection from the **Finish** to the corresponding **ResilientFinish** requires exchanging at least two signals between the two entities: 'publish' and 'release'. The publish signal is sent from **Finish** to the resilient store to create the resilient finish

object. The release signal flows in the other direction when the finish scope terminates (see the publish and release signals in Figure 3).

Recovering an async-finish control structure after a failure involves two main activities: 1) identifying orphaned tasks and adopting them by their grandparent finish, and 2) identifying tasks lost due to the failure and excluding them from the global count of pending activities. Uncertainties about the number and locations of orphaned tasks and the state of tasks spawned from a dead source complicate the above two activities. In the following sections we describe the different methods of dealing with these uncertainties in both the pessimistic finish protocol proposed by Cunningham et al. [2] and our optimistic protocol.

## 6.2 Resilient Pessimistic Finish

The pessimistic resilient finish protocol is built on the assumption that the resilient store is the only reliable component in the system, and that it should be capable of independently recovering the control structure of the computation. Independent recovery requires certainty about task and finish states, since seeking confirmations from other places is not permitted.

Task tracking is simpler in non-resilient finish compared to resilient finish. As described in Section 4, the resilient finish protocols need to eagerly notify task fork events before sending remote tasks to their destination. Pessimistic finish applies this method and expects to receive a ‘fork’ signal stating the source and destination of a potential task, and a ‘join’ signal stating the destination place at which the task terminated. It uses three variables to maintain task tracking information: `gc` to count pending tasks, `live[]` to count running tasks at each place, and `trans[][]` to count transiting tasks between any two places. See Listing 4 for a pseudocode of the pessimistic finish protocol.

### 6.2.1 Uncertainty about orphaned tasks

To accurately identify orphaned tasks, the pessimistic finish protocol requires each finish not only to publish itself in the store, but also to notify its parent resilient finish of its existence. Thus, in addition to ‘publish’ and ‘release’, pessimistic finish adds a third finish signal: ‘add\_child’. When a failure occurs, each resilient finish can identify which of its children need adoption without consulting other places. Depending on the resilient store implementation, it may be possible to merge the ‘publish’ and ‘add\_child’ signals in one message. Thus, pessimistic finish may require either two or three TD messages per finish. Adoption is performed by deactivating the adopted finish, and merging its task counts with the adopter’s counts. When a deactivated finish receives task signals, it forwards them to its adopter. The directive `ADOPTER_FORWARDING` in Listing 4 refers to this forwarding procedure.

### 6.2.2 Uncertainty about tasks spawned from a dead source

To avoid uncertainty about tasks spawned by a dead place, the pessimistic finish protocol drops all tasks transiting from a dead source place, even if they may be eventually received by their destination without any corruption. This may prevent valid tasks from making forward progress, however it enables the resilient finish object to independently recover the control structure. Assuming that `trans[s][*]` is the summation of transiting tasks from place `s` to any other place, when resilient finish detects the failure of place `s` it reduces the value of `gc` by the value of `trans[s][*]` and resets `trans[s][*]`. These updates exclude transiting tasks from the dead place from the scope of finish.

To maintain the consistency of the system, finish must also prevent the destination places from accepting tasks it excluded. That is achieved by introducing a third signal namely ‘toLive’ that the destination place uses to consult the resilient finish object on the validity of a received task. Resilient finish considers a task as valid only if both its source and destination are alive. In that case, it changes

the status of the task from transiting to live by decrementing `trans[s][d]` and incrementing `live[d]`, and acknowledges the validity of the task. Otherwise, it declares the task as invalid to direct the destination place to drop it. Note that the destination place cannot check the status of the source place locally, because its failure knowledge may be different from that of the resilient finish object.

In addition to excluding tasks transiting from and to dead places, pessimistic finish also excludes tasks living at dead places. On detecting a failure of a place `d`, the value `live[d]` is deducted from the global count, then is reset.

## 7 Our Proposed Protocol: Resilient Optimistic Finish

The main drawback of the pessimistic finish protocol is the assumption that all system components except the resilient store are unreliable, and that they cannot be used for recovery. Because of this, it requires the places to consult the resilient store before making any change in the task DAG to keep it synchronized with the state of tasks at all places.

In this section we will describe a message-optimal resilient termination detection protocol for the async-finish model. Our protocol makes the more practical assumption that a failure will impact a small minority of the system components [6, 8, 10], and that the majority of components will be available and capable of collaborating to recover the control structure when failures occur. We sacrificed some certainty of task status at the resilient store and empowered the places with more knowledge that can be used for resolving the uncertainty in case of a failure.

In a failure-free execution, the optimistic finish protocol uses only two finish signals: ‘publish’ and ‘release’; and two task signals: ‘fork’ and ‘join’. The resilient finish object keeps track of the number of pending tasks in its scope, but it cannot independently tell how many of these tasks are transiting and how many of them are running at their destinations. It uses a table of counters `transOrLive[][]` to record the number of tasks that may be in one of those states. When the source place `s` forks a task to the destination place `d`, `transOrLive[s][d]` and the global count `gc` are incremented. When the destination place receives the task, it *locally* determines whether or not the task is valid for execution – it does not consult the resilient finish object. If the task is valid, it executes it and sends a join signal when the task terminates. The join signal carries both the source and destination of the task, and result in decrementing `transOrLive[s][d]` and `gc`. On failure of a source place, the resilient finish object uses another table of counters `sent[][]` to resolve uncertainty about task status, as we will explain in Section 7.0.2.

### 7.0.1 Uncertainty about orphaned tasks

Identifying the child finishes that were lost due to a failure is key to identifying orphaned tasks. The pessimistic finish protocol uses the ‘add\_child’ signal to update a resilient finish object with the full list of children prior to the occurrence of any failure. In contrast, the optimistic finish protocol delays identifying the children worthy of adoption until a failure occurs. To achieve that, each resilient finish object records the id of its parent which it receives as part of the ‘publish’ signal.

The protocol relies on the fact that a child finish at place `d` will be created by one of the living tasks at place `d` governed by the parent finish. When a place `d` dies, each resilient finish object checks the value of `transOrLive[*][d]` to determine whether it has any pending tasks at that place. If there are no pending tasks at `d`, then there are no children worthy of adoption due to the failure of place `d`. Otherwise, it consults the resilient store itself to retrieve the list of children whose home place is `d`, and therefore require adoption. The parent finish records these children in a set called `ghosts`, and does not terminate until all of the ghosts have terminated. Thus termination is detected when `gc` reaches zero and the `ghosts` set is empty. See the condition of `tryRelease()` in Listing 5.

The reason why we refer to the adopted children as ghosts in this protocol is because we keep them active after their corresponding **finish** dies. The ghost finishes continue to govern their own tasks as normal, unlike the pessimistic finish protocol which deactivates the adopted children. When a ghost finish determines that all its tasks have terminated, it must send a 'term\_ghost' signal to its parent. When the parent receives this signal, it removes the child finish from its ghosts set.

## 7.0.2 Uncertainty about tasks spawned from a dead source

The optimistic protocol aims to empower each place to decide on the validity of received tasks *locally*, without consulting the resilient store. However, knowledge inconsistency may result in disagreements between the resilient store and the local finish object on the fate of tasks received from dead places. To keep the resilient store and the local finish objects in sync without adding overhead to failure-free execution, we require the resilient store to push its failure knowledge to the impacted local finish objects when failures occur. The local finish object saves this knowledge locally in a variable called `deny`, and uses it to determine the validity of received tasks. On receiving a task from place `p`, the destination place rejects the task if `deny[p]=true`. We added a new recovery signal called 'mark\_denied\_place' for that purpose.

*Excluding tasks to a dead place:* Tasks transiting or live at a destination place which has died can be safely excluded, since they are assumed to be lost with the place. After determining the ghost children, the resilient finish object can deduct `transOrLive[*][d]` from the global count, and reset `transOrLive[*][d]` for each dead place `d`. Any termination messages received from the dead place `d` must be discarded, otherwise they may incorrectly alter the global count.

*Excluding tasks from a dead place:* After a source place notifies the resilient store with its intention to fork a remote task to another place, the resilient store does not receive any more updates regarding the task until it terminates. If a failure hits the source place after the fork signal is processed, the resilient finish object cannot determine whether or not the source place had successfully sent the task. In other words, if `transOrLive[dead][d] == x`, how many of the `x` tasks were successfully received by place `d`? An optimistic resilient finish can only answer this question through communication with the destination place `d`. To resolve the uncertainty, we require each place to record the total number of received tasks from each source place `s` in a variable called `recv[s]`. We also require each resilient finish object to track the total number of tasks sent from each source place `s` to each destination place `d` in a variable called `sent[s][d]`. The difference between `sent[s][d]` and `recv[s]` at place `d` is the number of tasks that should be excluded. We use the same signal 'mark\_denied\_place' to calculate the above value and to stop place `d` from receiving future tasks from place `s`. On failure of the source place `s`, the resilient finish sends the id of place `s` and the value of `sent[s][d]` to the destination place `d`. The destination place marks the source as dead, and determines the number of tasks that should be excluded by comparing the number of tasks sent with the number received. It returns this value to the resilient finish object to adjust its task counters.

Handling the failure of both the source and the destination does not require additional rules. It is reduced to handling the failure of the destination, then handling the failure of the source.

## 7.0.3 Optimistic Finish TLA+ Specification

We developed a formal specification to model the above protocol using the Temporal Logic Actions tool (TLA+) and used it to test the correctness of the protocol. The model simulates all possible  $n$ -level task graphs that can be created on a  $p$ -place system, where each node of the task graph has at most two children. It simulates the failure of one place at any moment while the task graph is evolving, which is sufficient to verify correctness of the protocol in the case of loss of a source or destination place at any time. Testing was performed on an 8-core Intel i7-3770 3.40GHz system

running Ubuntu 14.04 operating system. It took a total of 2 hours and 59 minutes to verify correctness of all possible executions for 3-level task graphs on a 3-place system <sup>2</sup>.

## 8 Performance Evaluation

We conducted the following experiments on Raijin supercomputer at NCI, the Australian National Computing Infrastructure. Each compute node in Raijin has a dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors, and uses an Infiniband FDR network. We allocated 10 GiB of memory per node, statically bound each place to a separate core, and disabled hyperthreading.

### 8.1 Protocols Implementations with Different Resilient Stores

The resilient store is a key factor in the performance and the survivability of the implementation. We compared the performance of the pessimistic and optimistic protocol using two stores implemented natively within X10: a centralized store that saves all resilient finish objects at place-zero, and a distributed store that replicates each resilient finish object at two places (one replica is stored at the finish home along with the global finish object, and the second replica is created at the next place). The pessimistic finish protocol was evaluated using similar stores in [2], however, the distributed implementation was later removed from the source code repository due to its instability. We identified a serious bug in the replication protocol used and hence re-implemented a corrected version of the distributed store for both the pessimistic and optimistic protocols. Garbage collection in the optimistic finish implementations is performed by piggybacking GC requests to places in responses to their next finish signals <sup>3</sup> In the following results we use (P-p0) to refer to pessimistic place-zero, (O-p0) to refer to optimistic place-zero, (P-dist) to refer to pessimistic distributed, and (O-dist) to optimistic distributed.

### 8.2 BenchMicro

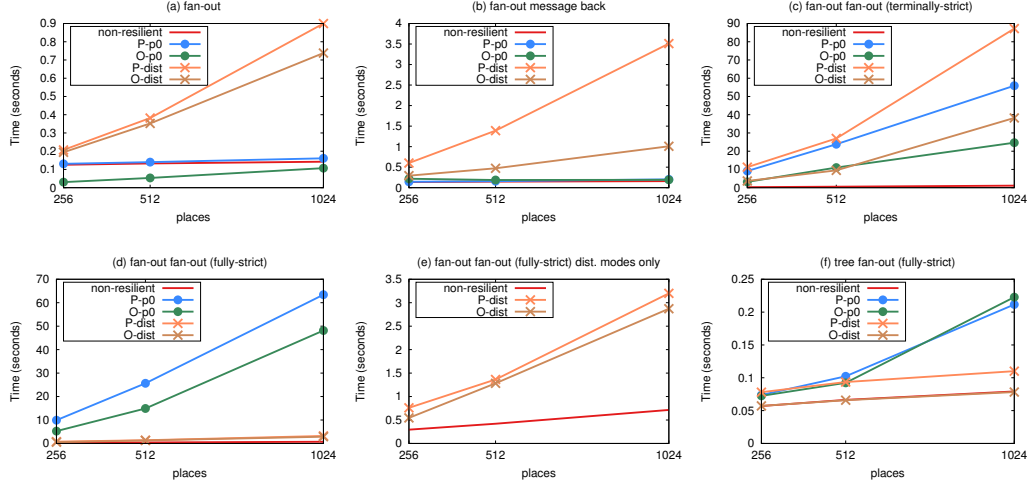
Cunningham et al. [2] designed the BenchMicro program to measure the overheads introduced by resilient finish in various distributed computational patterns, such as fan-out, all-to-all (or fan-out fan-out), and tree fan-out. We modified BenchMicro to start all the computations from the middle place, rather than from place-zero. This avoids giving an unfair advantage to the centralized implementations by allowing them to handle most of the signals locally. We recorded the average execution time for each pattern over 2 invocations of the BenchMicro program, in which the pattern is executed repeatedly over at least 10 seconds, with at least 10 repetitions. We report times for each place count (256, 512, 1024) in Figure 6. Our main conclusions from the results are as follows: 1) As expected, the optimistic protocols are successfully reducing the resilience overhead, and the percentage of reduction increases with the increase in the number of tasks. 2) Protocols that create big number of concurrent finishes, such as fan-out fan-out (fully-strict), and the tree fanout, scale poorly with a centralized store implementation, due to the sequential bottleneck of the store. For such computations, a distributed store is the most adequate option (see Figures 3-d,e,f).

### 8.3 LULESH

We now present results for LULESH shock hydrodynamics proxy application [9] with domain size  $30^3$ , running for 30 iterations. In the failure scenario, we kill place  $n/2$  just before iteration 15.

<sup>2</sup> Interested readers can check the specification and the outputs of verification in the public GitHub repository [12]

<sup>3</sup> The non-resilient finish in X10 adds explicit messages for garbage collection. GC not needed in pessimistic finish



■ **Figure 6** BenchMicro results. Figure d shows scaling for a fully-strict fan-out fan-out pattern, for all finish modes; figure e zooms in on the same results for distributed modes only.

LULESH involves point-to-point communication between neighboring places for exchanging ghost cells that uses distributed finish blocks. It also uses collectives, however, the collectives map directly to native MPI calls, hence do not use finish. Table 1 shows the performance results for LULESH over 64 and 1000 places. The initialization kernel of LULESH is highly communication-intensive, each place interacts with all its 26 neighbors to obtain global access to remote buffers used for ghost cell exchange. As shown in Table 1, the pessimistic modes result in higher overhead for this kernel compared to the optimistic modes. Overall, the optimistic mode achieves better performance results in both failure-free and failure scenarios.

■ **Table 1** LULESH performance. Times are in milliseconds. Overheads shown in parentheses.

	64 places			1024 places		
	init	step	total	init	step	total
Non-resilient	446	173	5652	1689	211	8184
P-p0	2190	172	7382 (31%)	10110	210	16581 (103%)
O-p0	1350	170	6500 (15%)	6963	209	13377 (63%)
P-dist	609	171	5754 (2%)	9928	223	17296 (111%)
O-dist	583	172	5767 (2%)	7735	219	14800 (81%)
With Failure P-p0	1848	172	10496 (86%)	9741	224	32277 (294%)
With Failure O-p0	1117	172	9193 (63%)	7164	211	26407 (223%)
With Failure P-dist	633	174	7905 (40%)	11072	226	47141 (476%)
With Failure O-dist	623	175	7923 (40%)	6541	252	33443 (309%)

## 9 Conclusion

We described *optimistic finish*, a termination detection protocol for the async-finish programming model. By reducing the signals required for tracking tasks and finish scopes, our protocol significantly



517 reduces the resilience overhead of computation-intensive applications and micro-benchmarks, and  
 518 enables them to reliably recover from failures.

#### ■ Listing 4 Pessimistic Finish Pseudocode

```

27 class P_ResilientFinish(id){
28   gc:int=0;
29   live:int[places];
30   trans:int[places][places];
31   allChildren:Set[Id];
32   adopter:Id;
33   def transit(src, dst){
34     ADOPTER_FORWARDING;
35     if (bothAlive(src, dst)){
36       trans[src][dst]++; gc++;
37     }
38   }
39   def toLive(src, dst){
40     ADOPTER_FORWARDING;
41     if (bothAlive(src, dst)){
42       trans[src][dst]--;
43       live[dst]++;
44       return true;
45     }
46     else return false;
47   }
48   def term(dst){
49     ADOPTER_FORWARDING;
50     if (!dst.isDead()){
51       live[dst]--; gc--;
52       if (gc == 0)
53         @F[id].release();
54     }
55   } };
56 class P_Finish(id) : Finish{
57   def make(parent:Finish) {
58     @RF[parent.id].addChild(id);
59     @store.publish(id);
60   } };
61 class P_LocFinish(id) : LocalFinish{
62   lc:int=0;
63
64   def fork(src, dst) {
65     @RF[id].transit(src, dst);
66   }
67   def begin(src, dst) {
68     val v = @RF[id].toLive(src,
69       dst);
70     if (v) { lc++; } return v;
71   }
72   def join(src, dst){
73     lc--;
74     if (lc == 0) @RF[id].term(dst);
75   } };

```

#### ■ Listing 5 Optimistic Finish Pseudocode

```

75 class OptResilientFinish(id){
76   gc:int=0; parent:Id;
77   transOrLive:int[places][places];
78   sent:int[places][places];
79   ghosts:Set[Id]; isGhost:bool;
80   def transit(src, dst){
81     if (bothAlive(src, dst)){
82       transOrLive[src][dst]++;
83       sent[src][dst]++; gc++;
84     }
85   }
86   def term(t, dst){
87     if (!dst.isDead()){
88       transOrLive[*][dst] -= t[*];
89       gc -= t[*]; tryRelease();
90     }
91   }
92   def termGhost(ghostId){
93     ghosts.remove(ghostId);
94     tryRelease();
95   }
96   def tryRelease(){
97     if (gc == 0 && ghosts.empty()){
98       if (isGhost)
99         @RF[parent].termGhost(id);
100     else @F[id].release();
101   }
102 } };
103 class O_Finish(id) : Finish{
104   def make(parent:Finish) {
105     //add_child signal not needed
106     @store.publish(id, parent.id);
107   } };
108 class O_LocFinish(id) : LocalFinish{
109   lc:int; deny:bool[places];
110   recv:int[places]; rep:int[places];
111   def fork(src, dst) {
112     @RF[id].transit(src, dst);
113   }
114   def begin(src, dst) {
115     if (deny[src]) return false;
116     lc++; recv[src]++; return true;
117   }
118   def join(src, dst){
119     lc--;
120     if (lc == 0) {
121       val t = recv - rep;
122       @RF[id].term(t, dst);
123     }
124   } };

```

---

References

---

- 1 Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- 2 David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: Efficient failure-aware programming. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pages 67–80, 2014.
- 3 Edsger W Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
- 4 Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2009.
- 5 Ten-Hwang Lai and Li-Fen Wu. An (n-1)-resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):63–78, 1995.
- 6 Jonathan Lifflander, Phil Miller, and Laxmikant Kale. Adoption protocols for fanout-optimal fault-tolerant termination detection. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13*. ACM, 2013.
- 7 Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.
- 8 Esteban Meneses, Xiang Ni, and Laxmikant V Kalé. A message-logging protocol for multicore systems. In *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 1–6. IEEE, 2012.
- 9 Josh Milthorpe, David Grove, Benjamin Herta, and Olivier Tardieu. Exploring the APGAS programming model using the LULESH proxy application. Technical Report RC25555, IBM Research, 2015.
- 10 Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *ACM/IEEE international conference for high performance computing, networking, storage and analysis (Supercomputing)*, pages 1–11. IEEE Computer Society, 2010.
- 11 Sara S. Hamouda. A forked GitHub repository of the X10 language with our optimistic finish implementations. <https://github.com/shamouda/x10/tree/optimistic>.
- 12 Sara S. Hamouda. TLA+ specification of the optimistic finish protocol and replication protocol. <https://github.com/shamouda/x10-formal-spec>.
- 13 Subbarayan Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, 1989.