Resilient Optimistic Termination Detection for the Async-Finish Model

Sara S. Hamouda[2][0000−0001−7300−9565]⋆ and Josh Milthorpe[1][0000−0002−3588−9896]

[1] Australian National University, Canberra, Australia
[2] INRIA, Paris, France
sara.hamouda@inria.fr,josh.milthorpe@anu.edu.au

Abstract. Driven by 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.
For distributed termination detection, maintaining a consistent view of task state across multiple unreliable processes requires additional book-keeping when creating or completing tasks and finish-scopes. Runtime systems which perform this book-keeping *pessimistically*, i.e. synchronously with task state changes, add a high communication overhead compared to non-resilient protocols. 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.
Performance results using micro-benchmarks and the LULESH hydrodynamics proxy application show significant reductions in resilience overhead with optimistic finish compared to pessimistic finish. Our optimistic finish protocol is applicable to any task-based runtime system offering automatic termination detection for dynamic graphs of non-migratable tasks.

Keywords: Async-finish · Termination detection · Resilience

1 Introduction

Recent advances in high-performance computing (HPC) systems have greatly increased parallelism, with both larger numbers of nodes, and larger core counts within each node. With 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 and ensuring resilience to component failures. As more industrial and scientific communities rely on HPC to drive innovation, there is a need for productive programming models for scalable resilient applications.

⋆ Work done while at the Australian National University, Canberra, Australia.

Many productive HPC programming models support nested task parallelism via composable task-parallel constructs, which simplify the expression of arbitrary task graphs to efficiently exploit available hardware parallelism. Termination detection (TD) – determining when all tasks in a subgraph are complete – is a key requirement for dynamic task graphs. In an unreliable system, additional work is required for correct termination detection in the presence of component failures. Task-based models for use in HPC must therefore support resilience through efficient fault-tolerant TD protocols.

The async-finish task model is a productive task parallelism model adopted by many asynchronous partitioned global address space (APGAS) languages. It represents a computation as a global task graph composed of nested sub-graphs, each managed by a `finish` construct. Finish embodies a TD protocol to track the termination of the asynchronous tasks spawned directly or transitively within its scope.

The first resilient TD protocol for the async-finish model was designed by Cunningham et al. [3] as part of the Resilient X10 project. Resilient X10 provides user-level fault tolerance support by extending the async-finish model with failure awareness. Failure awareness enables an application to be notified of process failures impacting the computation's task graph to adopt a suitable recovery procedure. Unsurprisingly, adding failure awareness to the async-finish model entails a cost; it requires the runtime system to perform additional book-keeping activities to correctly detect termination despite the gaps created in the computation's task graph.

Cunningham et al.'s TD protocol for Resilient X10 tracks all state transitions of remote tasks in order to maintain a consistent view of the computation's control flow. While this ensures a simple failure recovery process, it adds more termination signals than are strictly necessary during normal failure-free execution. Since it favors failure recovery over normal execution, we describe this protocol as 'pessimistic'.

In this paper, we review the pessimistic finish protocol, 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.

The remainder of the paper is organized as follows. Section 2 provides background on nested task parallelism and the X10 programming languages, and Section 3 presents related work. Section 4 proves the optimal number of messages required for correct async-finish termination detection. Section 5 describes the failure model and the challenges of recovering async-finish task graphs. Section 6 presents an abstract framework for implementing async-finish TD protocols. Based on this framework, we describe the pessimistic protocol in Section 7 and the optimistic protocol in Section 8. Section 9 describes a scalable resilient finish store. Section 10 presents the performance evaluations and Section 11 concludes.

2 Background

2.1 Nested Task Parallelism Models

Computations that entail nested termination scopes are generally classified as fully-strict or terminally-strict. Blumofe and Leiserson [2] describe a fully-strict task graph

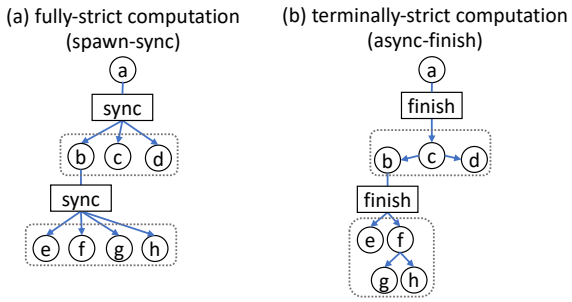

Fig. 1: Nested parallelism models. Dotted boxes are termination scopes; circles are tasks.

as one that has fork edges from a task to its children and join edges from each child to its direct parent. A task can only wait for other tasks it directly forked (see Figure 1-a). In contrast, a terminally-strict task graph 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 (see Figure 1-b). Cilk's spawn-sync programming model and X10's async-finish programming model are the most prominent representatives of fully-strict and terminally-strict computations, respectively. For dynamic irregular task trees, the async-finish model avoids unnecessary synchronization by relaxing the requirement to have each task to wait for its direct successors.

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. Although in this paper we focus on the async-finish model, the described resilient protocols are also applicable to the spawn-sync model.

2.2 The X10 Programming Model

We used the APGAS language X10 as a basis for our study of the termination detection protocols of the async-finish model. X10 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. A task is spawned at a particular place and cannot migrate to other places.

An X10 program dynamically generates an arbitrary task graph by nesting `async`, `at`, and `finish` constructs. The `async` construct spawns a new task at the current place. To spawn an asynchronous task at a remote place p, `at` is used with `async` as follows: (`at (p) async S;`). The `finish` construct is used for synchronization; it defines a scope of coherent tasks and waits for their termination. Each task belongs to one finish scope, and finish scopes can be nested. Exceptions thrown from any of the tasks are collected at the finish, and are wrapped in a `MultipleExceptions` object after finish terminates. 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();`).

3 Related Work

Dijkstra and Scholten (DS) [4] proposed one of the earliest and best-studied TD protocols for the so-called 'diffusing computation' model. In this 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 DS protocol is a *message-optimal* protocol such that for a computation that sends M basic messages, DS adds exactly M control messages to detect termination. It requires each intermediate process to signal its termination only after its successor processes terminate. This termination detection model is very similar to Cilk's fully-strict spawn-sync model. Fault-tolerant extensions of the DS algorithm are presented in [6, 7].

Lai and Wu [6] 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 or 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, which limits its applicability to large-scale HPC applications.

Lifflander et al. [7] took a practical approach for resilient TD of a fully-strict diffusing computation. Based on the assumption that multi-node failures are rare in practice, and that the probability of a k-node failure decreases as k increases, 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 identify its successors and their successors. Therefore, each process notifies its parent of its potential successor before activating it. Two protocols were proposed to address related failures, however, they cannot tolerate the failure of an interior (non-leaf) node and its parent.

To the best of our knowledge, the only prior work addressing resilient TD for the terminally-strict async-finish model was done in the context of the X10 language. Cunningham et al. [3] enhanced X10 to allow a program to detect the failure of a place through a `DeadPlaceException` from the **`finish`** constructs that control tasks running at that place. The TD protocol extends **`finish`** with an adoption mechanism that enables it to detect orphan tasks of its dead children and to wait for their termination before reporting a `DPE` to the application. This adoption mechanism ensures that failures will not leave behind hidden tasks at surviving places that may silently corrupt the application's state after recovery. However, it requires additional book-keeping activities for tracking the control flow of the computation, which results in high resilience overhead for failure-free execution. Our work extends [3] to provide a low-overhead resilient termination detection protocol for the async-finish model. Assuming failures are rare events, we designed a message-optimal 'optimistic' TD protocol that aims to optimize the failure-free execution performance of resilient applications.

Resilient distributed work-stealing runtime systems use fault tolerant protocols for tracking task migration under failure [5, 9]. Our work focuses on the APGAS model, in which tasks are explicitly assigned to places, hence they are not migratable.

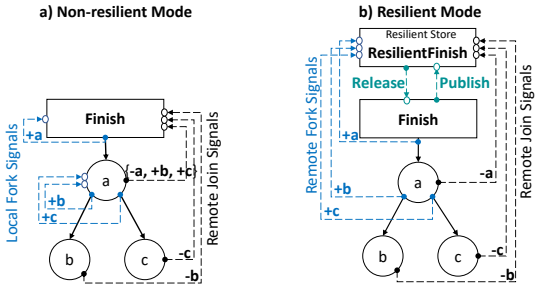

Fig. 2: Message-optimal async-finish TD protocols.

4 Message-Optimal Async-Finish Termination Detection

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

We assume a finish block which includes nested async statements that create distributed tasks, such that each task and its parent (task or finish) are located at different places. Messages sent to fork these tasks at their intended locations are referred to as *basic messages*. For example, in the task graphs in Figure 2, 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 [3] in which the TD metadata of the **`finish`** constructs are maintained safely in a *resilient store*. A **`finish`** and the resilient store exchange two signals: the `PUBLISH` signal is sent from the **`finish`** to the store to create a corresponding `ResilientFinish` object, and the `RELEASE` signal flows in the other direction when the finish scope terminates (see Figure 2-b).

As a finish scope evolves by existing tasks forking new tasks, finish needs to update its knowledge of the total number of active tasks so that it can wait for their termination. We refer to this number as the *global count* or `gc`. Finish forks the first task and sets `gc = 1`. A task must notify finish when it forks a successor task to allow finish to increase the number of active tasks (by incrementing `gc`). When a task terminates, it must notify finish to allow it to decrease the number of active tasks (by decrementing `gc`). When the last task terminates, `gc` reaches zero and finish is released. We use the terms `FORK` and `JOIN` to refer to the control signals used to notify finish when a new task is forked and when a running task terminates. Multiple signals from the same source may be packed in one message for better performance.

Lemma 1. *A correct non-resilient finish requires one TD control message per task.*

Proof. Finish detects termination only after all forked tasks terminate. Thus sending a `JOIN` signal when a task terminates is unavoidable for correct termination detection. During execution, a parent task may fork N successor tasks, and therefore, it must notify finish with N `FORK` signals for these tasks. Assuming that failures do not occur,

each task must eventually terminate and send its own `JOIN` signal. A task can buffer the `FORK` signals of its successor tasks locally and send them with its `JOIN` signal in the same message. Thus, with only one message per task, finish will eventually receive a `FORK` signal and a `JOIN` signal for each task, which guarantees correct termination detection.

Figure 2-a illustrates this method of non-resilient termination detection. When task `a` forks tasks `b` and `c`, it delays sending their `FORK` signals (`+b`, `+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`.

Lemma 2. *A correct resilient finish requires two TD control messages per task.*

Proof. In the presence of failures, tasks may fail at arbitrary times during execution. For correct termination detection, finish must be aware of the existence of each forked task. If a parent task fails in between forking a successor task and sending the `FORK` signal of this task to finish, finish will not track the successor task since it is not aware of its existence, and termination detection will be incorrect. Therefore, a parent task must eagerly send the `FORK` signal of a successor task before forking the task, and may not buffer the `FORK` signals locally. For correct termination detection, each task must also send a `JOIN` signal when it terminates. As a result, correct termination detection in the presence of failures requires two separate TD messages per task – a message for the task's `FORK` signal, and a message for the task's `JOIN` signal. The absence of either message makes termination detection incorrect.

Figure 2-b demonstrates this method of resilient termination detection, which ensures that a resilient finish is tracking every forked task. Assuming that a resilient finish can detect the failure of any node in the system, it can cancel forked tasks located at failed nodes to avoid waiting for them indefinitely. Note that in counting the messages, we do not consider the messages that the resilient store may generate internally to guarantee reliable storage of resilient finish objects. While a centralized resilient store may not introduce any additional communication, a replication-based store will introduce communication to replicate the objects consistently.

Lemma 3. *Optimistic resilient finish is a message-optimal TD protocol.*

Proof. Our proposed optimistic finish protocol (Section 8) uses exactly two messages per task to notify task forking and termination. Since both messages are necessary for correct termination detection, the optimistic finish protocol is message-optimal.

5 Async-Finish Termination Detection Under Failure

In this section, we use the following sample program to illustrate the challenges of async-finish TD under failure and the possible solutions. In Section 7 and Section 8, we describe how these challenges are addressed by the pessimistic protocol and the optimistic protocol, respectively.

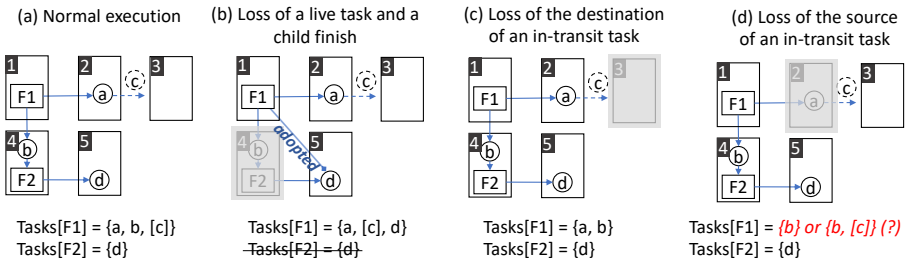

Fig. 3: Task tracking under failure. The square brackets mark in-transit tasks.

```
1 finish /*F1*/ {
2   at (p2) async { /*a*/ at (p3) async { /*c*/ } }
3   at (p4) async { /*b*/ finish /*F2*/ at (p5) async { /*d*/ } }
4 }
```

Failure model: We focus on process (i.e. place) fail-stop failures. A failed place permanently terminates, and its data and tasks are immediately lost. We assume that each place will eventually detect the failure of any other place, and that a corrupted message due to the failure of its source will be dropped either by the network module or the deserialization module of the destination place. We assume non-byzantine behavior.

Challenge 1 - Loss of termination detection metadata: As a computation evolves, finish objects are created at different places to maintain the TD metadata (e.g. the active tasks of each finish). Losing one of these objects impairs the control flow and prevents correct termination detection. To address this challenge, Cunningham et al. [3] proposed using a *resilient store* that can save the data reliably and survive 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. For the above program, we assume that `F1` and `F2` have corresponding resilient finish objects in the resilient store.

Challenge 2 - Adopting orphan tasks: When the finish home place fails, the finish may leave behind active tasks that require tracking. We refer to these tasks as orphan tasks. According to the semantics of the async-finish model, a parent finish can only terminate after its nested (children) finishes terminate. A parent finish can maintain this rule by adopting the orphan tasks of its dead children to wait for their termination. Figure 3-b shows the adoption of task `d` by `F1` after the home place of `F2` failed.

Challenge 3 - Loss of in-transit and live tasks: Each task has a source place and a destination (home) place, which are the same for locally generated tasks. The active (non-terminated) tasks of the computation can be either running at their home place (live tasks) or transiting from a source place towards their home place (in-transit tasks).

The *failure of the destination place* has the same impact on live and in-transit tasks. For both categories, the tasks are lost and their parent finish must exclude them from its global task count. For example, the failure of place 4 in Figure 3-b results in losing the live task `b`, and the failure of place 3 in Figure 3-c results in losing the in-transit task `c`, because its target place is no longer available.

Listing 1.1: Finish TD API.

```
1  abstract class Finish(id: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:Id) {
8    val gr:GlobalRef[Finish];
9    def fork(src, dst):void;
10   def join(src, dst):void;
11   def begin(src, dst):bool;
12 }
```

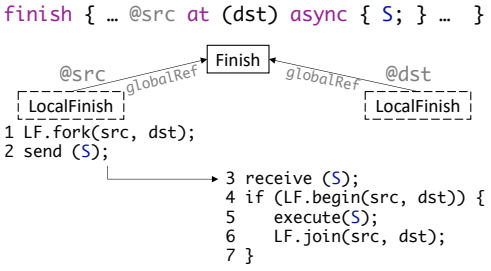

Fig. 4: Tracking remote task creation.

The *failure of the source place* has a different impact on live and in-transit tasks. Live tasks proceed normally regardless of the failure, because they already started execution at their destinations. However, in-transit tasks are more difficult to handle (Figure 3-d). Based on Lemma 2, in resilient mode, a source place must notify its finish of a potential remote task before sending the task to its destination. If the source place died after the finish received the notification, the finish cannot determine whether the potential task was: 1) never transmitted, 2) fully transmitted and will eventually be received by the destination, or 3) partially transmitted and will be dropped at the destination due to message corruption. A unified rule that allows finish to tolerate this uncertainty is to consider any in-transit task whose source place has died as a lost task and exclude it from the global task count. The finish must also direct the destination place to drop the task in case it is successfully received in the future.

To summarize, recovering the control flow requires the following: 1) adopting orphan tasks, 2) excluding live tasks whose destination place is dead, 3) excluding in-transit tasks whose source place or destination place is dead, and 4) preventing a destination place from executing an in-transit task whose source place is dead. The optimistic finish protocol achieves these goals using the optimal number of TD messages per task, unlike the pessimistic protocol which uses one additional message per task.

6 Distributed Task Tracking

In this section, we describe an abstract framework that can be used to implement termination detection protocols, based on the X10 runtime implementation. The essence of the framework is presented as pseudocode in Listing 1.1, and Figure 4. In Sections 6.3, 7, and 8, we will describe three termination detection protocols based on this framework.

6.1 Finish and LocalFinish Objects

A termination detection protocol is defined by providing concrete implementations of the abstract classes `Finish` and `LocalFinish` shown in Listing 1.1. For each **`finish`** block, an instance of `Finish` with a globally unique `id` is created to maintain a global view of the distributed task graph. When the task that created the finish reaches the

Listing 1.2: Non-resilient Finish

```
1  class NR_Finish(id) extends Finish {
2    gc:int=0; // global count
3    def terminate(live:int[places]) {
4      for (p in places) gc += live[p];
5      if (gc == 0) release();
6    }
7  }
8  class NR_LocalFinish(id) extends LocalFinish {
9    live:int[places]={0}; //signals buffer
10   def fork(src, dst) { live[dst]++; }
11   def begin(src, dst) { return true; }
12   def join(src, dst) { live[dst]--; @F[id].terminate(live); }
13 }
```

end of the block, it calls the function `wait` to block on the `latch` until all the tasks that were created by the finish block have terminated. When all tasks (direct and transitive) terminate, the runtime system calls the function `release` to release the blocked task. The runtime system links each finish object to its parent in a tree structure.

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 so that the `Finish` has an up-to-date view of the global control structure.

6.2 Task Events

`LocalFinish` defines three interfaces to track task events: `fork`, `begin`, and `join`. Figure 4 shows the invocation of the three task events when a source place `src` spawns a task at a destination place `dst`. On forking 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.

We describe the protocols 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 a remote `Finish` object, `LocalFinish` object and `ResilientFinish` object, respectively.

6.3 Non-Resilient Finish Protocol

Listing 1.2 describes a message-optimal implementation of a non-resilient finish. The finish object maintains a global count, `gc`, representing the number of active tasks. The `LocalFinish` maintains a `live` array to buffer the `FORK` and `JOIN` signal of its task and the `FORK` signals of successor tasks to other places. The `begin` event accepts all incoming tasks, because this non-resilient protocol is not prepared for receiving invalid tasks due to failures. The `join` event passes the signals buffer `live` to the finish object. Finish updates `gc` according to the passed signals and releases itself when `gc` reaches zero.

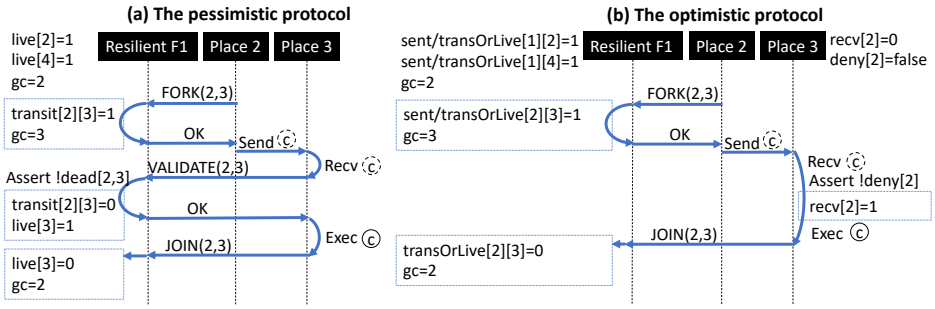

Fig. 5: The task tracking events as task c transitions from place 2 to place 3, based on Figure 3-a.

7 Resilient Pessimistic Finish

The pessimistic resilient finish protocol requires the resilient finish objects to track the tasks and *independently* repair their state when a failure occurs. Independent repair requires advance knowledge of the status of each active task (whether it is in-transit or live) and the set of children of each finish for adoption purposes.

Classifying active tasks into in-transit and live is necessary for failure recovery, because the two types of tasks are treated differently with respect to the failure of their source, as described in Section 5. Using only the `FORK` and the `JOIN` signals (see Section 4), a resilient finish can track a task as it transitions between the not-existing, active, and terminated states. However, these two signals are not sufficient to distinguish between in-transit or live tasks. The pessimistic protocol adds a third task signal that we call `VALIDATE` to perform this classification. Although the classification is only needed for recovery, the application pays the added communication cost even in failure-free executions.

The resilient finish object uses three variables for task tracking: `gc` to count the active tasks, `live[]` to count the live tasks at a certain place, and `trans[][]` to count the in-transit tasks between any two places. On receiving a `FORK` signal for a task moving from place `s` to place `d`, the resilient finish object increments the variable `trans[s][d]` and the global count `gc`. When the destination place receives a task, it sends a `VALIDATE` message to resilient finish to check if the task is valid for execution. If both the source and the destination of the task are active, resilient finish declares the task as valid and transforms it from the transit state to the live state. That is done by decrementing `trans[s][d]` and incrementing `live[d]`. On receiving a `JOIN` signal for a task that lived at place `d`, the resilient finish decrements the variables `live[d]` and `gc` (see Figure 5-a).

7.1 Adopting Orphan Tasks

Tracking the parental relation between finishes is key to identifying orphaned tasks. The pessimistic finish protocol requires each new finish not only to publish itself in the resilient store, but also to link itself to its parent. Thus, in addition to the `PUBLISH` and the `RELEASE` signals (see Section 4), a pessimistic finish uses a third signal `ADD_CHILD` to

connect a new resilient finish to its parent. When a parent finish adopts a child finish, it deactivates the resilient finish object of the child and adds the child's task counts to its own task counts. A deactivated finish forwards the received task signals to its adopter. The `FORWARD_TO_ADOPTER` directive in Listing 1.3 refers to this forwarding procedure.

7.2 Excluding Lost Tasks

When place `P` fails, the live tasks at `P` and the in-transit tasks from `P` and to `P` are considered lost. The number of lost tasks is the summation of `live[P]`, `trans[*][P]`, and `trans[P][*]`. After calculating the summation, the pessimistic finish object resets these counters and deducts the summation from the global count `gc` (see the `recover` method in Listing 1.3). If the source place of an in-transit task fails, the finish requests the destination place to drop the task using the response of the `VALIDATE` signal.

8 Our Proposed Protocol: Resilient Optimistic Finish

The optimistic finish protocol aims to provide reliable execution of async-finish computations using the minimum number of TD messages. It optimizes over the pessimistic protocol by removing from the critical path of task execution any communication that is needed only for failure recovery. In particular, it removes the `VALIDATE` signal which classifies active tasks into in-transit and live, and removes the `ADD_CHILD` signal which synchronously tracks the children of each finish. It compensates for the missing information due to removing these signals by empowering the places with additional metadata that can complete the knowledge of the resilient store at failure recovery time.

A resilient optimistic finish object uses the following variables for task tracking: `gc` to count the active tasks, `transOrLive[][]` to count the active tasks, which may be in-transit or live, given their source and destination, and `sent[][]` to count the total number of sent tasks between any two places, which includes active and terminated tasks. Each visited place within a finish scope records the following variables in its `LocalFinish` object: `recv[]` to count the number of received tasks from a certain place, and `deny[]` to check whether it can accept in-transit tasks from a certain place. Initially, tasks can be accepted from any place.

When a source place `s` forks a task to a destination place `d`, `transOrLive[s][d]`, `sent[s][d]` and the global count `gc` are incremented (see Listing 1.4-Line 32). When the destination place receives the task, it *locally* determines whether or not the task is valid for execution using its `deny` table (see Listing 1.4-Line 19). If the task is valid, the place executes it and sends a `JOIN` signal when the task terminates. The `JOIN` signal carries both the source and the destination of the task and results in decrementing `transOrLive[s][d]` and `gc` (see Figure 5-b). Note that `sent[][]` and `recv[]` are never decremented. We will show in Section 8.2 how the `sent[][]` and the `recv[]` tables are used for resolving the number of lost in-transit tasks due to the failure of their source.

8.1 Adopting Orphan Tasks

The optimistic protocol does not use the `ADD_CHILD` signal, but rather calculates the set of children needing adoption at failure recovery time.

Each resilient finish object records the id of its parent, which was given in the PUBLISH signal that created the object. The protocol relies on the fact that a child finish at place x will be created by one of the living tasks at place x governed by the parent finish. When a place P dies, each resilient finish object checks the value of transOrLive[*][P] to determine whether it has any active tasks at that place. If there are no active tasks at P, then there are no children needing adoption due to the failure of place P. Otherwise, it consults the resilient store to retrieve the list of children whose home place is P, and therefore require adoption. The parent finish records these children in a set called ghosts. Termination is detected when gc reaches zero and the ghosts set is empty (see the condition of tryRelease() in Listing 1.4). A valid resilient store implementation of the optimistic finish protocol must implement the FIND_CHILDREN function. This function is reduced to a search in a local set of resilient finish objects in a centralized resilient store, or a query to the backup of the dead place in a replication-based resilient store.

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 sends a REMOVE_CHILD signal to its parent (Line 50 in Listing 1.4). When the parent receives this signal, it removes the child finish from its ghosts set and checks for the possibility of releasing its corresponding finish.

8.2 Excluding Lost Tasks

Like the pessimistic protocol, we aim to exclude all transiting tasks from and to a dead place, and all live tasks at a dead place. However, because transiting and live tasks are not distinguished in our protocol, more work is required for identifying lost tasks.

For a destination place P, transOrLive[s][P] is the number of the in-transit tasks from s to P and the live tasks executing at P. If P failed, both categories of tasks are lost and must be excluded from the global count. After determining the ghost children (as described in Section 8.1), the resilient finish object can deduct transOrLive[*][P] from the global count, and reset transOrLive[*][P] for each failed place P. Any termination messages received from the dead place P must be discarded, otherwise they may incorrectly alter the global count. Handling the failure of both the source and the destination reduces to handling the failure of the destination.

For a source place P, transOrLive[P][d] is the number of the in-transit tasks from P to d and the live tasks sent by P and are executing at d. If P failed, only the in-transit tasks are lost and must be excluded from the global count; the live tasks proceed normally. An optimistic resilient finish can only identify the number of in-transit tasks through communication with the destination place d. Place d records the total number of received tasks from P in recv[P]. At the same time, the resilient finish object records the total number of sent tasks from P to d in sent[P][d]. The difference between sent[P][d] and recv[P] is the number of transiting tasks from P to d. The resilient finish object relies on a new signal COUNT_TRANSIT to calculate this difference and to stop place d from receiving future tasks from place P by setting deny[P] = **true** (see the COUNT_TRANSIT method in Listing 1.4, and its call in Listing 1.4-Line 64).

8.3 Optimistic Finish TLA Specification

TLA (Temporal Logic of Actions) [10] is a specification language for documentation and automatic verification of software systems. The system's specification includes an initial state, a set of actions that can update the system's state, and a set of safety and liveness properties that describe the correctness constraints of the system. The TLA model checker tool, named TLC, tests all possible combinations of actions and reports any detected violations of the system's properties.

We developed a formal model for the optimistic finish protocol using TLA to verify the protocol's correctness. Using 22 TLA actions, the model can simulate 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 c children. It can also simulate the occurrence of one or more place failures as the task graph evolves. The model specification is available at [11].

The distributed TLC tool currently cannot validate liveness properties, such as 'the system must eventually terminate', which we needed to guarantee in our protocol. Using the centralized TLC tool, it was infeasible for us to simulate large graph structures without getting out-of-memory errors due to the large number of actions in our model. Therefore, we decided to use a small graph configuration that can simulate all scenarios of our optimistic protocol. In order to verify the case when a parent finish adopts the tasks of a dead child, we need at least a 3-level graph, such that the finish at the top level can adopt the tasks at the third level that belong to a lost finish at the second level. In our protocol, separate cases handle the failure of the source place of a task, and the failure of the destination place of a task. With one place failure we can simulate either case. The case when a task loses both its source and destination requires killing two places, however, in our protocol handling the failure of both the source and destination is equivalent to handling the failure of the destination alone. Therefore, one place failure is sufficient to verify all rules of our protocol. Because we use the top finish to detect the full termination of the graph, we do not kill the place of the top finish. Therefore, we need two places or more in order to test the correctness of the failure scenarios.

Testing was performed using TLC version 1.5.6 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 the correctness of our protocol over a 3-level task tree with a branching factor of 2 using 3 places, with one place failure at any point in the execution.

9 Finish Resilient Store Implementations

Cunningham et al. [3] described three resilient store implementations, of which only two are suitable for HPC environments. One is a centralized store that holds all resilient finish objects at place-zero, assuming that it will survive all failures. The centralized nature of this store makes it a performance bottleneck for large numbers of concurrent finish objects and tasks. The other store is a distributed store that replicates each finish object at two places – the home place of the finish, which holds the master replica and the next place, which holds a backup replica. Unfortunately, this scalable implementation was later removed from the code base of Resilient X10 due to its complexity and instability. As a result, users of Resilient X10 are currently limited to using the non-scalable centralized place-zero finish store for HPC simulations.

Listing 1.3: Pessimistic Finish.

```
abstract class P_ResilientStore {
  def PUBLISH(id):void;
  def ADD_CHILD(parentId, childId):void;
}
class P_Finish(id:Id) extends Finish {
  def make(parent:Finish) {
    @store.ADD_CHILD(parent.id, id);
    @store.PUBLISH(id);
  }
}
class P_LocalFinish(id:Id) extends
     LocalFinish {
  def fork(src, dst) {
    @RF[id].FORK(src, dst);
  }
  def join(src, dst){
    @RF[id].JOIN(src, dst);
  }
  def begin(src, dst) {
    return @RF[id].VALIDATE(src, dst);
  }
}
class P_ResilientFinish(id:Id) {
  gc:int=0;
  live:int[places];
  trans:int[places][places];
  children:Set[Id];
  adopter:Id;
  def FORK(src, dst){
    FORWARD_TO_ADOPTER;
    if (bothAlive(src, dst)) {
      trans[src][dst]++; gc++;
    }
  }
  def JOIN(src, dst) {
    FORWARD_TO_ADOPTER;
    if (!dst.isDead()) {
      live[dst]--; gc--;
      if (gc == 0) @F[id].release();
    }
  }
  def VALIDATE(src, dst) {
    FORWARD_TO_ADOPTER;
    if (bothAlive(src, dst)) {
      trans[src][dst]--;
      live[dst]++;
      return true;
    }
    else return false;
  }
  def addChild(cId) {
    children.add(cId);
  }
  def recover(dead) {
    // adopt orphaned tasks
    for (c in children) {
      if (c.home == dead) {
        trans += @RF[c].trans;
        live += @RF[c].live;
        gc += @RF[c].gc;
        @RF[c].adopter = id;
      }
    }
    // exclude lost tasks
    gc -= trans[dead][*] + trans[*][dead]
     + live[dead];
    trans[dead][*] = 0;
    trans[*][dead] = 0;
    live[dead] = 0;
    if (gc == 0) @F[id].release();
  }
}
```

Listing 1.4: Optimistic Finish.

```
abstract class O_ResilientStore {
  def PUBLISH(id, parentId):void;
  def FIND_CHILDREN(id, place):Set[Id];
}
class O_Finish(id:Id) extends Finish {
  def make(parent:Finish) {
    @store.PUBLISH(id, parent.id);
  }
}
class O_LocalFinish(id:Id) extends
     LocalFinish {
  deny:bool[places]; recv:int[places];
  def fork(src, dst) {
    @RF[id].FORK(src, dst);
  }
  def join(src, dst){
    @RF[id].JOIN(src, dst);
  }
  def begin(src, dst) {
    if (deny[src]) return false;
    else { recv[src]++; return true; }
  }
  def COUNT_TRANSIT(nSent, dead) {
    deny[dead] = true;
    return nSent - recv[dead];
  }
}
class O_ResilientFinish(id:Id) {
  gc:int=0; parent:Id;
  transOrLive:int[places][places];
  sent:int[places][places];
  ghosts:Set[Id]; isGhost:bool;
  def FORK(src, dst){
    if (bothAlive(src, dst)){
      transOrLive[src][dst]++; gc++;
      sent[src][dst]++;
    }
  }
  def JOIN(src, dst){
    if (!dst.isDead()){
      transOrLive[src][dst]--; gc--;
      tryRelease();
    }
  }
  def removeChild(ghostId) {
    ghosts.remove(ghostId); tryRelease();
  }
  def tryRelease() {
    if (gc == 0 && ghosts.empty())
      if (isGhost)
        @RF[parent].removeChild(id);
      else @F[id].release();
  }
  def recover(dead) {
    if (transOrLive[*][dead] > 0) {
      val c = @store.FIND_CHILDREN(id,
     dead);
      ghosts.addAll(c);
      for (g in c) @RF[g].isGhost = true;
    }
    gc -= transOrLive[*][dead];
    transOrLive[*][dead] = 0;
    for (p in places) {
      if ( transOrLive[dead][p] > 0) {
        val s = sent[dead][p];
        val t = @LF[id].COUNT_TRANSIT(s,
     dead);
        transOrLive[dead][p] -= t;
        gc -= t;
      }
    }
    tryRelease();
  }
}
```

9.1 Reviving the Distributed Finish Store

Because a centralized place-zero finish store can significantly limit the performance of Resilient X10, we decided to reimplement a distributed finish store for Resilient X10 for both optimistic and pessimistic protocols. Using TLA's model checker, we identified a serious bug in the replication protocol described in [3] for synchronizing the master and the backup replicas of a finish. The problem in their implementation is that the master replica is in charge of forwarding task signals to the backup replica on behalf of the tasks. If the master dies, a task handles this failure by sending its signal directly to the backup. In cases when the master fails after forwarding the signal to the backup, the backup receives the same signal twice – one time from the dead master and one time from the task itself. This mistake corrupts the task counters at the backup and results in incorrect termination detection.

Using TLA, we designed a replication protocol (available in [11]) that requires each task to communicate directly with the master and the backup. The protocol ensures that each signal will be processed only once by each replica in failure-free and failure scenarios. When one replica detects the failure of the other replica, it recreates the lost replica on another place using its state. The protocol ensures that if both replicas are lost before a recovery is performed, the active tasks will reliably detect this catastrophic failure, which should lead the Resilient X10 runtime to terminate. Otherwise, the distributed store can successfully handle failures of multiple unrelated places. Because the failure of place-zero is unrecoverable in the X10 runtime system, our distributed finish implementations do not replicate the finish constructs of place zero.

10 Performance Evaluation

We conducted experiments on the 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 statically bound each place to a separate core. The X10 compiler and runtime were built from source revision `36ca628` of the `optimistic` branch of our repository `https://github.com/shamouda/x10.git`, which is based on release 2.6.1 of the X10 language. The experiments use the Native (C++ backend) version of X10 compiled using gcc 4.4.7 and use MPI-ULFM [1] for inter-place communication. MPI-ULFM is a fault tolerant MPI implementation that provides efficient fault tolerant point-to-point and collective communication interfaces over Infiniband and other networks. We built MPI-ULFM from revision `e87f595` of the `master` branch of the repository `https://bitbucket.org/icldistcomp/ulfm2.git`.

10.1 Microbenchmarks

Cunningham et al. [3] designed the BenchMicro program to measure the overhead 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 patterns from the middle place, rather than from place-zero. This avoids giving an unfair

Table 1: Slowdown factor versus non-resilient finish with 1024 places. Slowdown factor = (time resilient / time non-resilient). The "Opt. %" columns show the percentage of performance improvement credited to the optimistic finish protocol.

Pattern	Finish count	Tasks/Finish	**P-p0**	**O-p0**	**Opt. %**	**P-dist**	**O-dist**	**Opt. %**
1 Fan out	1	1024	2.3	0.9	**59%**	9.2	7.9	**14%**
2 Fan out message back	1	2048	1.4	1.2	**15%**	22.3	7.2	**68%**
3 Fan out fan out	1	1024^2	51.4	23.9	**53%**	95.6	39.2	**59%**
4 Fan out fan out with nested finish	1025	1024	90.9	80.8	**11%**	4.1	3.8	**7%**
5 Binary tree fan out	512	2	8.6	8.5	**2%**	1.4	1.1	**27%**
6 Synchronous ring around	1024	1	1.8	1.7	**1%**	1.8	1.8	**0%**

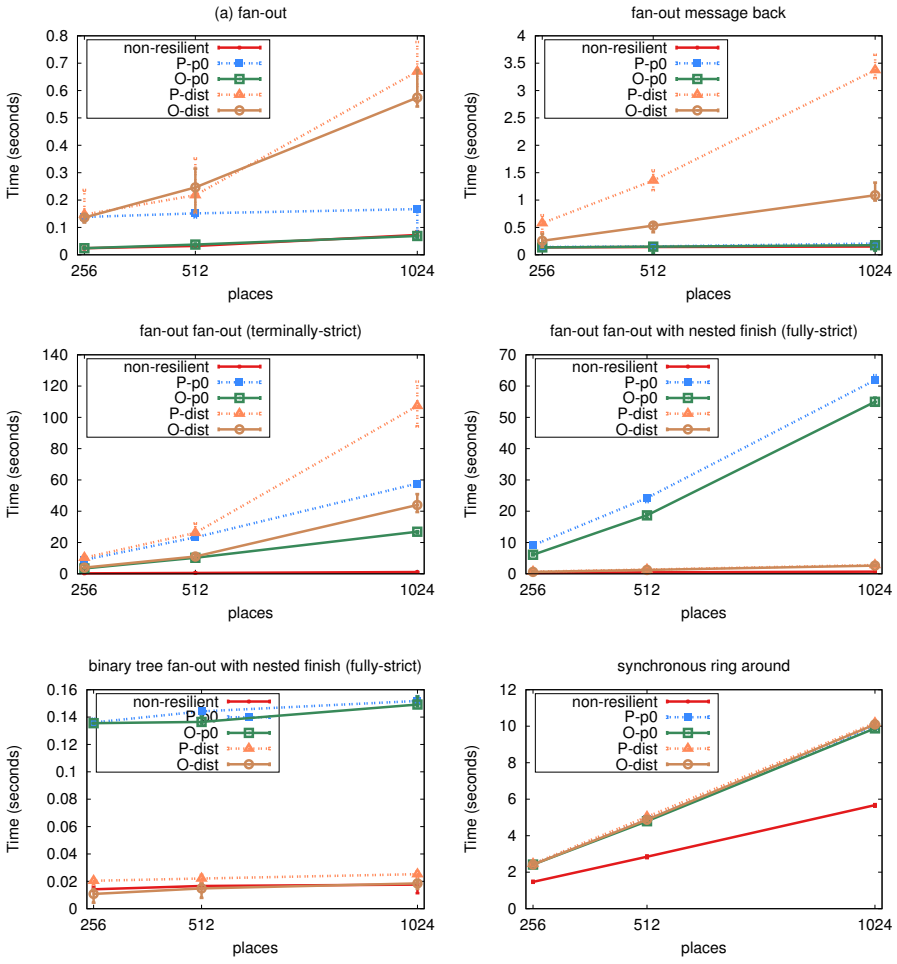

Fig. 6: BenchMicro results.

advantage to the centralized implementations by allowing them to handle most of the signals locally.

We measured the time to execute each pattern using 256, 512 and 1024 places, with one place per core. Each configuration was executed 30 times. Figure 6 shows the median with error bars representing the range between the 25th percentile, and the 75th percentile. **P-p0**, **O-p0**, **P-dist**, and **O-dist** refer to pessimistic-centralized, optimistic-centralized, pessimistic-distributed and optimistic-distributed implementations, respectively. Table 1 summarizes the performance results with 1024 places.

From the results, we observe: 1) our proposed optimistic protocol reduces the resilience overhead of the async-finish model for all patterns. 2) the improvement with the optimistic protocol is greater as the number of remote tasks managed by a finish increases. 3) the more concurrent and distributed the finish scopes are in the program, the greater the improvement observed with the resilient distributed finish implementations.

10.2 LULESH

X10 provides a resilient implementation of the LULESH shock hydrodynamics proxy application [8] based on rollback-recovery. It is an iterative SPMD application that executes a series of stencil computations on an evenly-partitioned grid and exchanges ghost regions between neighboring places at each step.

We measured the time to execute LULESH with domain size $(30 \times N)^3$ for 60 iterations. In resilient modes, checkpointing is performed every 10 iterations. In the failure scenarios, we start three spare places and kill a victim place every 20 iterations – exactly at iterations 5, 35, and 55. Therefore, a total of 75 iterations are executed, because each failure causes the application to re-execute 5 iterations. The victim places are $N/4$, $N/2$, and $3N/4$, where N is the number of places. Both failure and checkpoint rates are chosen to be orders of magnitude higher than would be expected in a real HPC system, to allow checkpoint and recovery costs to be accurately measured. Table 2 and Figure 7 show the weak scaling performance results using different TD implementations.

LULESH uses the fan-out finish pattern multiple times for creating the application's distributed data structures and for spawning a coarse-grain task at each place to compute on the local domain. These remote tasks do not add a resilience overhead, because the fan-out finishes start from place zero. The initialization kernel is highly communication-intensive – each place interacts with all its 26 neighbors to obtain access to remote buffers used for ghost cell exchange. This kernel is re-executed after each failure to reinitialize the references to the ghost regions. The optimistic finish protocol is highly effective in reducing the resilience overhead of this kernel and speeding up recovery.

Each LULESH step performs point-to-point communication between neighboring places for exchanging ghost regions, as well as collective functions. However, the collectives map directly to native MPI-ULFM calls, hence do not use finish. Ghost exchange is performed using finish blocks that manage a small number of tasks. With 1000 places, the measured resilience overhead of a single step is: 13% for P-p0, 8% for O-p0, 10% for P-dist, and only 4% for O-dist.

The application applies in-memory checkpointing by saving a copy of its state locally and another copy at the next place. Each copy is saved within a finish block that controls a single remote task; hence the advantage of the optimistic protocol is minimal.

Table 2: Average execution time for different LULESH kernels (times in seconds, finish resilience overhead shown in parentheses).

Places	Mode	Init.		Step		Ckpt	Detect.	Reinit.	Total time 60 steps (0 ckpt+0 fail)		Total time 60 steps (6 ckpt+0 fail)	Total time 75 steps (6 ckpt+3 fail)
343	non-res.	0.97		0.073					5.33			
	P-p0	6.22	(539%)	0.091	(26%)	0.16	0.57	7.21	11.70	(119%)	12.64	37.35
	O-p0	4.15	(326%)	0.092	(27%)	0.15	0.47	5.12	9.69	(82%)	10.58	28.75
	P-dist	2.02	(107%)	0.085	(17%)	0.05	0.23	1.93	7.12	(33%)	7.44	15.20
	O-dist	1.36	(39%)	0.082	(12%)	0.05	0.12	1.40	6.26	(17%)	6.58	12.36
1000	non-res.	1.72		0.085					6.82			
	P-p0	10.01	(482%)	0.096	(13%)	0.20	1.28	10.68	15.75	(131%)	16.92	54.25
	O-p0	7.49	(335%)	0.092	(8%)	0.18	0.66	8.54	12.99	(90%)	14.09	43.07
	P-dist	2.50	(45%)	0.094	(10%)	0.06	0.53	2.55	8.12	(19%)	8.46	19.07
	O-dist	2.41	(40%)	0.089	(4%)	0.06	0.36	2.50	7.73	(13%)	8.08	18.01

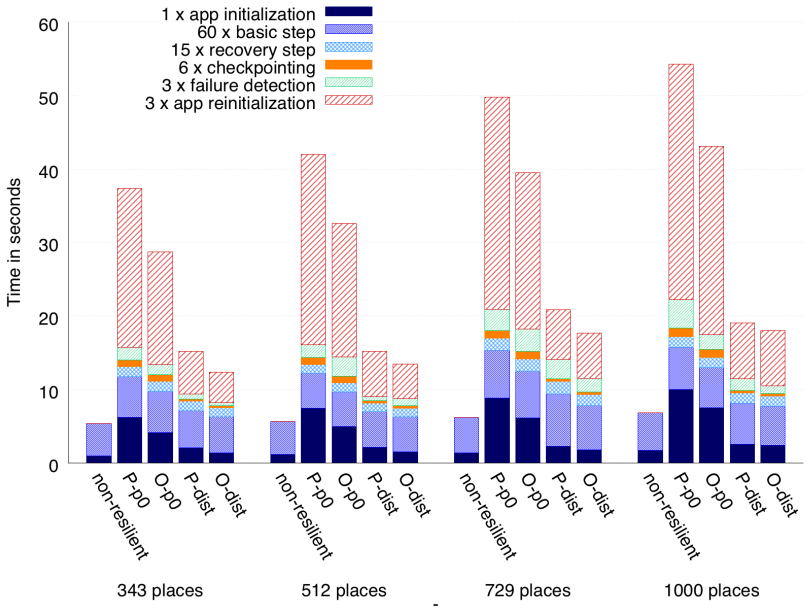

Fig. 7: LULESH weak scaling performance (1 core per place; requires n^3 places)

The reported failure detection time is the time between killing a place and the time when the fan-out finish that controls the execution of the algorithm reports a `DeadPlaceException`. This occurs only after all its tasks terminate with errors due to the failure of the victim (global failure propagation is achieved by ULFM's communicator invalidation mechanism). Reducing the runtime's tracking activities for termination detection accelerates task processing as well as failure detection using optimistic finish.

The computational pattern of LULESH is widely represented in the HPC domain. Overall, the optimistic finish protocol is successful in reducing the resilience overhead

of the application in failure-free and failure scenarios. The frequent use of concurrent finish scopes demonstrates the scalability advantage of the distributed finish store.

11 Conclusion

We described *optimistic finish*, a resilient message-optimal termination detection protocol for the productive task model, async-finish. By reducing the signals required for tracking tasks and finish scopes, our protocol significantly reduces the resilience overhead of overly decomposed parallel computations and enables them to reliably recover from failures.

Acknowledgements

We would like to thank David Grove and Olivier Tardieu of IBM T. J. Watson Research Center and Peter Strazdins of the Australian National University for their valuable comments on this work. The research used resources from the National Computational Infrastructure and NECTAR cloud, which are supported by the Australian Government.

References

1. Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., Dongarra, J.J.: An evaluation of user-level failure mitigation support in MPI. In: EuroMPI'12. pp. 193–203 (2012)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM **46**(5), 720–748 (1999)
3. Cunningham, D., Grove, D., Herta, B., Iyengar, A., Kawachiya, K., Murata, H., Saraswat, V., Takeuchi, M., Tardieu, O.: Resilient X10: Efficient failure-aware programming. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 67–80 (2014)
4. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters **11**(1), 1–4 (1980)
5. Kestor, G., Krishnamoorthy, S., Ma, W.: Localized fault recovery for nested fork-join programs. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 397–408. IEEE (2017)
6. Lai, T.H., Wu, L.F.: An (n-1)-resilient algorithm for distributed termination detection. IEEE Transactions on Parallel and Distributed Systems **6**(1), 63–78 (1995)
7. Lifflander, J., Miller, P., Kale, L.: Adoption protocols for fanout-optimal fault-tolerant termination detection. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM (2013)
8. Milthorpe, J., Grove, D., Herta, B., Tardieu, O.: Exploring the APGAS programming model using the LULESH proxy application. Tech. Rep. RC25555, IBM Research (2015)
9. Stewart, R., Maier, P., Trinder, P.: Transparent fault tolerance for scalable functional computation. Journal of Functional Programming **26** (2016)
10. The TLA Home Page. `http://lamport.azurewebsites.net/tla/tla.html`
11. TLA+ specification of the optimistic finish protocol and the replication protocol. `https://github.com/shamouda/x10-formal-spec`