Resilience in High-Level Parallel Programming Languages

Sara S. Hamouda

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

June 2019

Except where otherwise indicated, this thesis is my own original work.

Sara S. Hamouda
13 October 2018

To my parents, Hoda and Salem.

Acknowledgements

I grant all my gratitude to Almighty Allah, for blessing me with this wonderful learning journey and giving me the strength to complete it.

Crossing the finish line would not have been possible without the great support and assistance of a group of wonderful people. First of all, my supervisors, Josh Milthorpe, Peter Strazdins and Steve Blackburn, who offered their ultimate guidance and support to remove every obstacle in the way and enable me to achieve my goals. I wish to express my deepest appreciation to my primary supervisor, Josh Milthorpe, who dedicated many hours for discussing my work, refining my ideas, connecting me to collaborators, and challenging me with difficult questions sometimes to help me learn more. Thank you, Josh, for all the favors you showed me during these years. Without your encouragement and generous support, this thesis would not have been possible.

I am very grateful to the X10 team; David Grove, Olivier Tardieu, Vijay Saraswat, Benjamin Herta, Louis Mandel, and Arun Iyengar, for giving me the opportunity to collaborate with them in the Resilient X10 project, and for the invaluable experience I gained from this collaboration. I am also very grateful to the computer systems group in the research school of computer science at the ANU for the valuable feedback and recommendations given during the PhD monitoring sessions. I would like to thank J. Eliot B. Moss for valuable discussions about transactional memory systems during his visit to the ANU, and Peter Strazdins for valuable discussions that led to the design of the resilience taxonomy described in this thesis.

I am grateful to the Australian National University for providing me the HDR merit and stipend scholarship. This research was undertaken with the assistance of resources and services from the National Computational Infrastructure (NCI), which is supported by the Australian Government.

Finally, I would to thank my family and friends for the continuous encouragement and the happy moments they favored me with. My nephews, Nour Eldeen, Omar, and Eyad, are now too young to read these words, but I hope one day they will open my thesis and read this line to know how much I love them and how much chatting with them on a regular basis was my main source of happiness during my remote study years. To my dear family; Hoda, Salem, Mai, Ahmed and Esraa, thanks for supporting me every step of the way.

Abstract

The consistent trends of increasing core counts and decreasing mean-time-to-failure in supercomputers make supporting task parallelism and resilience a necessity in HPC programming models. Given the complexity of managing multi-threaded distributed execution in the presence of failures, there is a critical need for task-parallel abstractions that simplify writing efficient, modular, and understandable fault-tolerant applications.

MPI User-Level Failure Mitigation (MPI-ULFM) is an emerging fault-tolerant specification of MPI. It supports failure detection by returning special error codes and provides new interfaces for failure mitigation. Unfortunately, the unstructured form of failure reporting provided by MPI-ULFM hinders the composability and the clarity of the fault-tolerant programs. The low-level programming model of MPI and the simplistic failure reporting mechanism adopted by MPI-ULFM make MPI-ULFM more suitable as a low-level communication layer for resilient high-level languages, rather than a direct programming model for application development.

The asynchronous partitioned global address space model is a high-level programming model designed to improve the productivity of developing large-scale applications. It represents a computation as a global control flow of nested parallel tasks that use global data partitioned among processes. Recent advances in the APGAS model supported control flow recovery by adding failure awareness to the nested parallelism model — async-finish — and by providing structured failure reporting through exceptions. Unfortunately, the current implementation of the resilient async-finish model results in a high performance overhead that can restrict the scalability of applications. Moreover, the lack of data resilience support limits the productivity of the model as it shifts the challenges of handling data availability and atomicity under failure to the programmer.

In this thesis, we demonstrate that resilient APGAS languages can achieve scalable performance under failure by exploiting fault tolerance features in emerging communication libraries such as MPI-ULFM. We propose multi-resolution resilience, in which high-level resilient constructs are composed from efficient lower-level resilient constructs, as an approach for bridging the gap between the efficiency of user-level fault tolerance and the productivity of system-level fault tolerance. To address the limited resilience efficiency of the async-finish model, we propose 'optimistic finish' — a message-optimal resilient termination detection protocol for the finish construct. To improve programmer productivity, we augment the APGAS model with resilient

data stores that can simplify preserving critical application data in the presence of failure. In addition, we propose the 'transactional finish' construct as a productive mechanism for handling atomic updates on resilient data. Finally, we demonstrate the multi-resolution resilience approach by designing high-level resilient application frameworks based on the async-finish model.

We implemented the above enhancements in the X10 language, an embodiment of the APGAS model, and performed empirical evaluation for the performance of resilient X10 using micro-benchmarks and a suite of transactional and non-transactional resilient applications. Concepts of the APGAS model are realized in multiple programming languages, which can benefit from the conceptual and technical contributions of this thesis. The presented empirical evaluation results will aid future comparisons with other resilient programming models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis studies the challenges of supporting productive and efficient resilience support in high-level parallel programming languages. In particular, it considers the Asynchronous Partitioned Global Address Space (APGAS) programming model [Saraswat et al., 2010] exemplified by the X10 programming language [Charles et al., 2005].

1.1 Background and Motivation

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 [Schroeder and Gibson, 2007; Bergman et al., 2008; Cappello et al., 2009; Shalf et al., 2010; Ferreira et al., 2011; Dongarra et al., 2011]. Programmers of HPC systems must therefore address the twin challenges of efficiently exploiting available parallelism, while ensuring resilience to component failures. As more industrial and scientific communities rely on HPC as a driving power for their innovation, the need for productive programming models that simplify the development of scalable resilient applications continues to grow.

The Message Passing Interface (MPI), the de facto standard programming model for HPC applications, is carefully designed for performance scalability rather than resilience or productivity. It does not provide native task-parallel abstractions; however, users handle this limitation by integrating MPI programs with other tasking libraries at the expense of increased programming effort and integration complexities [Chamberlain et al., 2007; Hayashi et al., 2017]. In light of the criticality of failures on large-scale computations, the MPI Fault Tolerance Working Group (FTWG) was formed to study different proposals for extending MPI with fault tolerance interfaces. After developing three proposals, FT-MPI [Fagg and Dongarra, 2000], Run Through Stabilization (RTS) [Hursey et al., 2011], and MPI User Level Failure Mitigation (MPI-ULFM) [Bland et al., 2012b], the MPI FTWG has converged towards the MPI-ULFM specification.

MPI-ULFM is currently the only active proposal for adding fault tolerance semantics to the coming MPI-4 standard. It enables applications to recover from fail-stop process failures by providing failure detection and failure mitigation interfaces in MPI. Failure reporting is done by returning special error codes from a subset of MPI interfaces. To avoid altering the whole code with conditional checking for errors, a cleaner option for handling failures is by registering a global error handling function. However, using a global error handler isolates failure recovery from the piece of code where the failure was detected, which complicates failure identification and recovery [Laguna et al., 2016; Fang et al., 2016][1]. This simplistic unstructured form of failure reporting harms the modularity and the clarity of MPI-ULFM programs and complicates composing multiple fault tolerance techniques. Because of the low-level programming model of MPI and the above programmability challenges of MPI-ULFM, we argue that MPI-ULFM can be more useful as a low-level communication base for resilient high-level programming models rather than a direct programming model for application development.

The difficulties involved in developing distributed multi-threaded applications using low-level programming models, such as MPI, motivated the development of productivity-oriented programming models for HPC. The most influential project in that direction is DARPA's High Productivity Computing Systems project, which resulted in the emergence of the APGAS programming model and its earliest examples: X10 [Charles et al., 2005] and Chapel [Chamberlain et al., 2007].

The APGAS model avoids the main productivity barriers of MPI — a fragmented memory model and lack of task parallelism support — by providing a global view for distributed memory and supporting nested task parallelism. The async-finish task model emerged in APGAS languages as a more flexible nested parallelism model than the fork-join model [Guo et al., 2009]. The words async and fork represent constructs for spawning asynchronous tasks, and the words finish and join represent synchronization constructs. While finish can track asynchronous tasks spawned directly or transitively by the current task, join can only track asynchronous tasks spawned directly by the current task. Hence, the async-finish model can express not only fork-join task graphs but also other task graphs with arbitrary synchronization patterns. Unfortunately, most of the research on APGAS programming models has focused on its performance and productivity and ignored issues related to resilience.

To address this limitation, the X10 team has recently developed Resilient X10 (RX10), which extends the APGAS model with user-level fault tolerance [Cunningham et al., 2014; Crafa et al., 2014]. Fault tolerance is added by extending async-finish with failure awareness and structured failure reporting through exceptions. A resilient finish can detect the loss of any of its tasks due to a process failure and consequently raises an exception. Unlike MPI-ULFM, adopting a structured form of failure reporting in RX10 enables adding fault tolerance to existing codes in a clear and understandable way. It also facilitates the hierarchical composition of fault tolerance strategies. For example, see Listing 1.1.

[1] See [Laguna et al., 2016] for a more detailed description of this issue using a sample code.

Listing 1.1: Structured failure handling using the resilient async-finish model. A place represents an operating system process in X10. The statement `async S;` spawns an asynchronous task to execute `S`. The statement `at (p) S;` executes `S` at place `p`. The statement `finish S;` blocks the current task until all asynchronous tasks spawned by `S` terminate. The lines with a gray background are those added for fault tolerance.

```
1  def local_iteration(i:Int) {
2      try {
3          finish compute_using_neighbor();
4      } catch (ex:DeadPlaceException) { // neighbor died
5          //local recovery using approximation
6          val success = estimate_without_neighbor();
7          if (!success) throw new ApproximationFailure();
8      }
9  }
10 def global_iteration(i:Int) {
11     try {
12         finish for (p in places) at (p) async {
13             local_iteration(i);
14         }
15         checkpoint();
16     } catch(ex:ApproximationFailure) {
17         //restart this iteration
18         i--;
19         //global recovery using checkpoint/restart
20         val success = load_last_checkpoint();
21         if (!success) throw new FatalException();
22     }
23 }
24 def main () {
25     try {
26         for (var i:Int = 1; i < 10; i++) {
27             global_iteration(i);
28         }
29     } catch(ex:FatalException) {
30         print("fatal error occurred");
31     }
32 }
```

Listing 1.1 is a pseudocode for an iterative algorithm that uses two nested `finish` scopes to govern the execution of each iteration. The outer `finish` at Line 12 governs the execution of an iteration across all the processes. Failures detected by this finish are handled aggressively by restarting the entire iteration (Lines 16–22). Failing to restart is a fatal error (Lines 29–31). The inner `finish` at Line 3 governs the computation assigned to a single process at each iteration. Each process calculates a result using its neighboring process. This algorithm attempts to recover from a neighbor's failure locally by generating an approximate result (Lines 4–8) rather than restarting the iteration. It raises an exception to the outer `finish` only when the approximation algorithm fails.

Despite the advantages of the resilient async-finish model, RX10 has had major practicality issues. First, adding failure awareness while maintaining the flexibility of the async-finish model added a significant performance overhead to common application patterns. Second, RX10 did not leverage recent fault tolerant MPI implementations for inter-process communication, which limited its portability to supercomputer platforms. Finally, RX10 shifted the whole burden of failure handling to programmers, as it did not provide any support for commonly used fault tolerance techniques or any mechanism for protecting application data. We address these issues in this thesis aiming to provide a productive efficient programming model for developing resilient HPC applications.

1.2 Multi-Resolution Resilience

When designing resilient applications, the choice of the resilience approach directly impacts performance and productivity. System-level resilience provides fault tolerance to applications transparently using a generic fault tolerance technique. This approach is attractive for productivity; however, no fault tolerance technique is universally efficient for all applications. On the other hand, user-level resilience allows users to customize fault tolerance to specific application characteristics. This approach is attractive for performance; however, it incurs a high productivity cost.

We argue that a balance between the efficiency of user-level fault tolerance and the productivity of system-level fault tolerance can be achieved by supporting *multi-resolution resilience*. Multi-resolution resilience is a special type of user-level resilience that requires the programming model to provide efficient and composable resilient constructs that can be used for building resilient frameworks at different levels of abstraction. We recognize that both efficiency and composability are necessary features for a practical implementation of multi-resolution resilience. The absence of efficiency harms performance, while the absence of composability harms productivity as it complicates building higher-level frameworks.

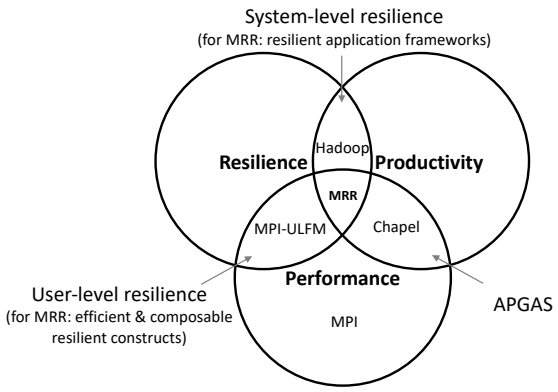

Figure 1.1: APGAS multi-resolution resilience (MRR) for reconciling performance, resilience, and productivity.

Figure 1.1 shows how multi-resolution resilience can enable APGAS languages to reconcile performance, resilience, and productivity. The Venn diagram highlights that each of the priorities of APGAS, user-level resilience, and system-level resilience is concerned with only two of the three features. For APGAS, reconciling performance and productivity is a key design objective. In addition, composable synchronization constructs are already available for supporting nested task parallelism. By extending these constructs with failure awareness, APGAS languages can gain the performance advantages of user-level resilience. They can also gain the productivity advantage of system-level resilience by using these constructs for building resilient application frameworks that handle fault tolerance on behalf of programmers. However, realizing this ambitious goal is challenging. Taking RX10 as an example implementation of multi-resolution resilience, we investigate performance and productivity problems that can hinder achieving this goal. We describe these problems in the next section.

1.3 Problem Statement

The composable async-finish model is a suitable base for multi-resolution resilience. However, adding failure awareness to the async-finish model does not come for free. It requires the runtime system to perform additional book-keeping activities for tracking the control flow of the computation. Our early experience with RX10 demonstrated that the composability of the async-finish model did not result in performance efficiency due to the high resilience overhead imposed by the book-keeping activities of the runtime system [Hamouda et al., 2015]. The main source of the overhead is due to the use of a resilient termination detection protocol for the `finish` construct that pessimistically tracks every state transition of remote tasks. In this thesis, we investigate the possibility of designing an optimistic finish protocol that reduces the resilience overhead in failure-free executions by allowing some uncertainty about the states of remote tasks. Such a protocol must be able to correctly resolve any uncertainties at failure recovery time.

Although an optimistic finish protocol may succeed in reducing the resilience overhead in failure-free scenarios, it cannot eliminate the resilience overhead entirely. In this thesis, we investigate the possibility of eliminating the resilience overhead for certain classes of applications by exploiting resilience capabilities in emerging fault-tolerant message-passing libraries. In particular, we focus on the MPI-ULFM library and evaluate its suitability to the RX10 programming model. To the best of our knowledge, our work is the first to evaluate MPI-ULFM in the context of a high-level parallel programming language.

Initial development of RX10 [Cunningham et al., 2014] focused mainly on recovering the computation's control flow and left the burden of protecting application data to the programmer. The complexities of ensuring data availability and consistency in the presence of failures is exacerbated in applications that require atomic updates on distributed data. In this thesis, we address the challenge of supporting efficient data resilience support in APGAS languages, including support for distributed transactions.

1.4 Research Questions

The objective of this thesis has been to answer the following questions:

- How to improve the resilience efficiency of the async-finish task model?
- How to exploit the fault tolerance capabilities of MPI-ULFM to improve the scalability of resilient APGAS languages?
- How to improve the productivity of resilient APGAS languages that support user-level fault tolerance?

1.5 Thesis Statement

High-level programming models can bridge the gap between resilience, productivity and performance by supporting multi-resolution resilience through efficient and composable resilient abstractions.

1.6 Contributions

This thesis focuses on software mechanisms for tolerating fail-stop process failures. It contributes multiple enhancements to the X10 language to provide practical support for multi-resolution resilience based on the async-finish task model. The following are the main contributions of my thesis:

- A taxonomy of resilient programming models.
- A detailed semantic mapping between X10's resilient asynchronous execution model and ULFM's failure reporting and recovery semantics.
- A novel message-optimal resilient termination detection protocol, named 'optimistic finish', that improves the efficiency of the resilient finish construct.
- A novel extension to the async-finish model to support resilient transactions that simplify handling distributed data atomicity in non-resilient and resilient applications.
- Extending RX10 with data resilience support by designing two resilient stores that shift the burden of handling data availability and consistency from the programmer to the runtime system and standard library.
- Demonstrating the multi-resolution resilience approach by building two productive resilient application frameworks based on the async-finish model. The first framework is a resilient iterative framework for bulk-synchronous applications, and the second framework is a parallel workers framework suitable for embarrassingly parallel applications.

- An empirical performance evaluation of the above enhancements using micro-benchmarks and applications with up to 1024 cores.

Contributions of the thesis are presented in the following publications: [Hamouda et al., 2015], [Hamouda et al., 2016], [Grove et al., 2019], and [Hamouda and Milthorpe, 2019].

1.7 Thesis Outline

The thesis follows the following structure:

- Chapter 2 describes a taxonomy of resilience characteristics, provides a broad overview of resilience features in different programming models, and describes the X10 programming model in detail.
- Chapter 3 describes the details of the integration between MPI-ULFM and RX10 and evaluates the resulting performance optimizations (related publication: [Hamouda et al., 2016]).
- Chapter 4 describes the optimistic finish protocol — our proposed protocol for improving the efficiency of resilient nested parallelism in APGAS languages — and evaluates the protocol in different computation patterns (related publication [Hamouda and Milthorpe, 2019]).
- Chapter 5 describes multiple extensions to X10 that aim at reducing the complexity of handling data resilience in X10 applications. It describes a proposed transactional finish construct for handling distributed data atomicity. It also describes multiple productive application frameworks. The value of these frameworks is demonstrated by developing a suite of transactional and non-transactional resilient applications. The chapter concludes with a detailed analysis of the performance of these applications with up to 1024 cores (related publications: [Hamouda et al., 2015], [Hamouda et al., 2016], [Grove et al., 2019]).
- Chapter 6 describes the thesis conclusions and directions for future work.

Chapter 2

Background and Related Work

In this chapter, we review the foundational concepts of resilience and their implementation in a wide range of programming models. The objective of our review is to identify not only the common approaches but also the neglected approaches in the area of resilience research. We also aim to identify the most suitable programming model for providing multi-resolution resilience — our proposed approach to user-level resilience which aims to combine productivity and performance scalability.

After defining resilience in Section 2.1, we describe a taxonomy of resilient programming models in Section 2.2. In Section 2.3, we use the taxonomy as a basis for describing the resilience characteristics of a wide range of programming models. The conclusions derived from our review are summarized in Section 2.3.6. Because we use X10 as a basis for our research, we dedicate Section 2.4 to describing the X10 programming model and its resilience features in detail.

2.1 Resilience Definition

Although resilience is sometimes used as a synonym for fault tolerance, there are other broader definitions. Almeida et al. [2013] reviewed a number of definitions of resilience and concluded with this quite inclusive definition:

> [resilience is] the emerging property of a system that is effective and efficient in accommodating to and recovering from changes in a broad sense with minimal impact on the quality of the service provided (avoiding failures as much as possible and performing as close as possible to its defined goals). [Almeida et al., 2013]

Almeida et al. consider fault tolerance as a necessary but not a sufficient property for resilience, because the impact on performance after recovery is an important factor that should not be ignored. They also consider resilience as a dynamic system property that requires the system to be able to recognize changes and adapt to them.

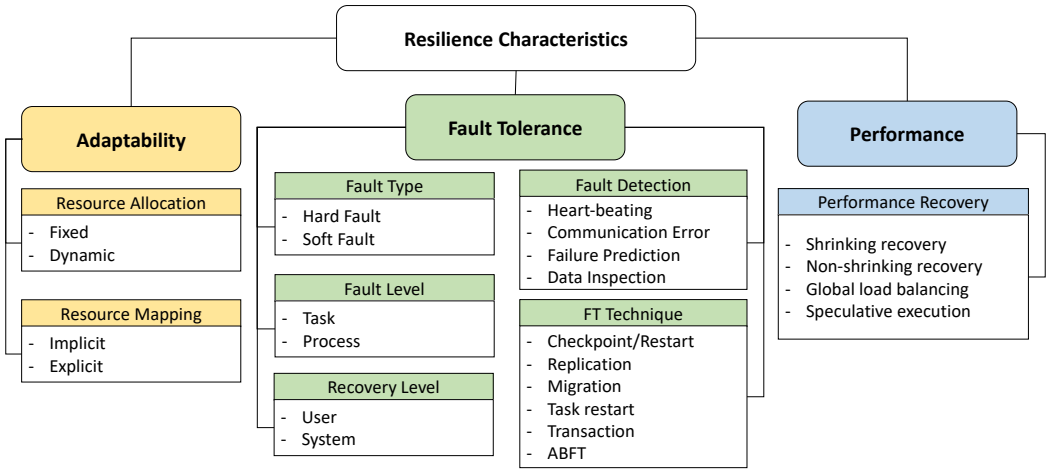

Figure 2.1: Taxonomy of resilient programming models.

2.2 Taxonomy of Resilient Programming Models

Based on Almeida et al.'s definition, we developed a taxonomy of resilient programming models that considers features in three broad categories: adaptability, fault tolerance, and performance management. Although the literature is rich with reviews for the different fault tolerance approaches and their implementations [Elnozahy et al., 2002; Cappello, 2009; Maloney and Goscinski, 2009; Egwutuoha et al., 2013; Bland et al., 2013], our review is unique in classifying programming models according to features in all three categories. The taxonomy framework enables us to zoom-in on the internal implementations of the different programming models and to identify missing properties in certain programming models that can provide more efficient support for resilience. This section provides a brief description of the taxonomy presented in Figure 2.1. The two features in our taxonomy, resource mapping and fault level, are adapted from the taxonomy in [Thoman et al., 2017].

2.2.1 Adaptability

A system's adaptability describes its ability to dynamically adjust the execution environment in response to resource failure. The taxonomy dissects adaptability into two properties: resource allocation and resource mapping.

2.2.1.1 Resource Allocation

This property describes the allocation policy of the hardware resources (i.e. compute nodes or processors) for executing the application processes.

Fixed Allocation

A runtime system in this category allocates a fixed-size pool of resources for the application. It requires specifying the number of processes at startup time and does not provide the capability of starting new processes during execution. When a process fails, the application can only use the remaining processes to continue processing. Some applications can easily adapt to a shrinking set of resources by redistributing the workload among the remaining resources. In contrast, it is challenging for applications that use static workload partitioning to tolerate the loss of resources. These applications may compensate the fixed allocation policy by allocating spare processes in advance to be used for recovery. However, this method results in wasting resources as the spare processes may remain idle for most or all of the processing time.

Dynamic Allocation

A runtime system in this category is capable of expanding its resource allocation during execution. Rather than allocating spare resources in advance, the runtime system can dynamically allocate new resources to compensate for failed ones. In programming models in which the processes are tightly coupled, such as MPI and APGAS, dynamic process allocation may impose global synchronization on the applications to reach a consistent view on the available set of processes. If a globally consistent view is not required, the runtime system can employ an asynchronous handshaking protocol that eventually propagates the identity of the new processes to the rest of the world.

2.2.1.2 Resource Mapping

In distributed computing, coarse-grained execution units (i.e. processes) collaborate on processing the computation's work units. The resource mapping property describes the level of user control over mapping the work units to the execution units. It also implies the level of coupling between the logical work units and the hardware resources that support them. The term "work unit" maps to different concepts in different programming models. For example, it maps to an actor in the Charm++ programming model, an entire process in MPI, and a task in task-based programming models. In the following, we explain how the resource mapping policy is crucial in determining the resilience model a programming model can support.

Explicit Mapping

A programming model in this category gives the programmer full control over mapping the work units to the processes. With this control, programmers are capable of implementing data and work placement strategies that minimize communication and improve the scalability of their applications. They can also implement exact or approximate mechanisms for recovering lost data given their knowledge of the role each process is playing in evaluating the computation. Programming models that

require explicit resource mapping are therefore flexible to supporting generic as well as application-specific fault tolerance techniques.

Implicit Mapping

Programming models in this category decouple the work units from the physical resources that execute them by hiding coarse-grain parallelism from the programming abstraction. The programmer expresses the computation as a graph of process-independent work units and relies on the runtime system to implicitly map the work units to the available processes to achieve the best performance. The runtime system can employ locality-aware scheduling or enable the programmer to provide locality hints [Mattson et al., 2016] to improve its decisions regarding work placement. Because the work-to-process mapping knowledge is mainly owned by the runtime system rather than the programmer, recovering from process failure may only be feasible at the runtime level. Consequently, programming models that provide implicit resource mapping are often not adequate for user-level recovery and the application of algorithmic-based fault tolerance techniques for handling process failures.

2.2.2 Fault Tolerance

The fault tolerance part of the taxonomy covers the types of faults, the common fault detection mechanisms, the common fault tolerance techniques, and the different fault levels and recovery levels that runtime systems can support.

2.2.2.1 Fault Type

Faults are generally classified into hard faults or soft faults. The frequency of both fault types is expected to rise on future exascale supercomputers. That is due to the increasing parallelism and transistor density on these systems and the reducing redundancy and component voltage for power saving purposes [Feng et al., 2010].

Hard Faults

A hard fault occurs when a certain component unexpectedly stops operating; the resulting failures are called fail-stop failures. A crashed process due to a hardware or a software error is an example of a fail-stop failure. Fail-stop failures are the most common type of failures in distributed systems [Du et al., 2012].

Soft Faults

A soft fault occurs due to bit-flips corrupting data in disk, memory or processor registers [Cao et al., 2015]. It impacts both shared memory systems and distributed memory systems. Sources of soft faults include electrical noise, temperature fluctuations, and electrostatic discharge [Du et al., 2011; Mukherjee et al., 2005]. Unlike hard faults, soft faults do not necessarily interrupt the execution of the program which

makes them more difficult to detect. If not detected, they can lead the impacted programs to silently generate wrong results or eventually terminate catastrophically.

2.2.2.2 Fault Level

In the single-core processor era, parallel processing was mainly achieved by executing multiple sequential processes in parallel at different nodes. The advent of multi-core and heterogeneous architectures demanded finer granularity of parallelism at the application level to utilize the available hardware parallelism more efficiently. Nowadays, a typical parallel application is expected to execute multiple tasks in parallel within each process by mapping the tasks to the available CPUs and accelerators in the node. Adding new levels of parallelism introduces new levels of faults that the application can experience. In this category, we classify the faults that a runtime system can recover into task faults and process faults.

Task Faults

Individual tasks within a process may fail due to soft faults, hard faults, or programming errors. The process in which the task executes is responsible for detecting and recovering the failed task.

Process Faults

An entire process can also fail, resulting in immediate loss of the data and the tasks it owns. Other processes communicating with the failed process can detect its failure and trigger the recovery procedure.

2.2.2.3 Recovery Level

This property classifies the resilient programming models according to the level at which faults are handled. The two main categories are system-level fault tolerance and user-level fault tolerance.

System Level

System-level fault tolerance refers to embedding an application-oblivious failure recovery mechanism in the runtime which can be leveraged by applications transparently or semi-transparently. It is favored for programming productivity since it shifts the burden of handling failures away from the user. However, adopting a one-size-fits-all approach for handling failures may impose expensive overheads on certain applications that can take advantage of algorithmic-based fault tolerance techniques.

User Level

User-level fault tolerance refers to runtimes that do not provide a built-in recovery mechanism. Instead, the runtime provides failure detection and reporting services

that users rely on to implement application-specific recovery mechanisms. The main advantage of user-level fault tolerance over system-level fault tolerance is that the former allows users to tailor more efficient recovery methods based on their expert knowledge of the underlying algorithmic properties.

2.2.2.4 Fault Detection

In this property, we describe the common fault detection techniques applied in distributed programming models.

Heartbeat

Heartbeating is an active failure detection mechanism. A process observes the status of another process by exchanging periodic heartbeat messages with this process. A process is considered alive as long as it continues acknowledging the heartbeat messages, otherwise, it is considered dead.

Communication Errors

In this mechanism, a process is considered dead when communication actions with it fail to complete successfully. It is a passive mechanism for failure detection.

External Notifications

This category describes runtime systems that rely on external components to detect failures and notify the live processes about them. For example, some distributed systems create a daemon process at each node to monitor and propagate the status of the application processes, other systems use third-party Reliability, Availability, and Serviceability (RAS) services to achieve the same goal.

Failure Prediction

Proactive fault tolerance techniques, such as process migration, rely on failure predictors to notify the runtime system of possible failures. When an imminent fault is detected, the predictor sends a warning signal to the runtime system to enable it to take failure avoidance actions. Failure prediction models are often designed based the history of failures on a particular system using its RAS log files [Gainaru et al., 2012; Liang et al., 2006; Sahoo et al., 2002].

Data Inspection

Inspection of the application data during execution can reveal the impact of soft faults. Some algorithms are designed to have intrinsic properties that impose well-defined constraints on the data values [Rizzi et al., 2016]; violations of these constraints indicate the occurrence of a soft error. For example, adding checksum vectors to original application data is a common mechanism for linear algebra applications to

detect and even correct data corruption errors. A more general approach for data inspection is by using computation replication. By invoking multiple instances of the same computation, data corruption can be detected by comparing the results of the different instances [Ni et al., 2013].

2.2.2.5 Fault Tolerance Technique

In the following, we describe the common fault tolerance techniques used for handling soft faults and hard faults.

Checkpoint/Restart

Checkpoint/Restart is the most widely used fault tolerance technique in HPC. It prepares the application to face failures by capturing snapshots of the application state during execution. Checkpoint/restart is a rollback-recovery mechanism — when failures occur, the application is restarted from an old state using the latest saved checkpoint. In a distributed execution, capturing a consistent state across several processes in the absence of a global clock is challenging. Many protocols have been investigated in the literature for tackling this challenge [Elnozahy et al., 2002; Maloney and Goscinski, 2009], where the protocols are generally classified into: coordinated protocols and uncoordinated protocols.

Coordinated checkpointing: In a coordinated protocol, checkpointing is performed as a collective operation between all the processes. At the start of a checkpointing phase, each process pauses executing application actions, completes any pending communications with the other processes, then saves its local state to a reliable storage. Because application-level communications are stopped, checkpointing does not require capturing the communication state between the processes. A failure of one or more processes requires restarting all the processes from the latest checkpoint.

Uncoordinated checkpointing with message logging: An uncoordinated protocol avoids synchronizing the processes globally for checkpointing or restarting. Each process decides the checkpointing time independently from the other processes. Upon a failure, only the failed process is restarted using its latest checkpoint. In order to bring the restarted process to a state that is consistent with the rest of the processes, the other processes replay the communication events that were performed since the checkpoint time of the restarting process. Because of that, most uncoordinated checkpointing protocols require the processes to log the communication messages and to store them as part of their checkpoint.

Replication

Process replication is a forward-recovery mechanism. It is based on the intuitive idea of executing multiple instances of the same computation in parallel, so that when one instance is impacted by a failure another instance can proceed towards normal completion. The main challenge of this procedure is ensuring that the different instances remain identical. This goal can be achieved using atomic broadcast protocols

that ensure that changes to the application state are applied on all of the replicas in the correct order. The main drawback of process replication is introducing high resource utilization overhead. Despite that, replication is being investigated as a more practical fault tolerance technique for exascale systems than checkpoint/restart [Bougeret et al., 2014; Ropars et al., 2015]. Some studies suggest that global checkpoint/restart may not be feasible on exascale systems because checkpointing the state of an exascale machine is likely to exceed its Mean Time Between Failures (MTBF) [Cappello et al., 2009; Dongarra et al., 2011].

In addition to replicating processes, replication may also be performed for fine-grained work units (e.g. actors, tasks) for failure recovery or load balancing.

Migration

Migration [Wang et al., 2008; Meneses et al., 2014] is a proactive fault tolerance technique that aims to avoid failures by predicting imminent faults and migrating the work units (e.g. actors, tasks or processes) from nodes that are likely to fail to other healthier nodes. The accuracy of the prediction model used is a determining factor for the performance overhead imposed by migration and for the level of protection it can deliver to applications. False positives lead to unnecessary migrations, while false negatives lead to crashed executions. To mitigate for false negatives, migration is often used with another fault tolerance technique, such as checkpoint/restart [Egwutuoha et al., 2013]. The ability to predict some of the failures enables the application to checkpoint its state less frequently, thereby enhancing the overall execution performance.

Transactions

A distributed transaction provides atomic execution of multiple distributed actions, such that either all the actions take effect (i.e. the transaction commits) or none of them do (i.e. the transaction aborts). Atomic commit protocols are employed behind the scenes to track the issued actions, acquire the needed locks, detect conflicts with other transactions, and finalize the transaction properly by either committing or aborting it. Handling process failure during transaction execution has been widely studied by the database community [Skeen, 1981; Bernstein and Goodman, 1984].

Resilient transactions can simplify the development of fault tolerant applications as they remove the burden of data consistency and failure recovery from the user to the transactional memory system. Most HPC programming models lack support for distributed transactions due to the scalability limitations of the distributed commit protocols [Harding et al., 2017]. These protocols require multiple phases of communication between all the transaction participants which incurs high performance overhead.

Task Restart

Task restart is a popular fault tolerance mechanism for dataflow systems that are represented by various big-data systems [Isard et al., 2007; Dean and Ghemawat, 2008; Zaharia et al., 2010] and HPC task-based runtime systems [Aiken et al., 2014; Mattson et al., 2016]. In these systems, the computation is expressed as a graph of inter-dependent tasks. The dependencies between the tasks are explicitly defined at the program level and passed to the runtime system. Process failures result in losing tasks and data objects that other tasks may depend on. From the task-dependence graph, the runtime system can identify and re-execute the set of tasks that can regenerate the lost data in order to satisfy the dependencies of the pending tasks to direct the execution towards successful termination. Checkpointing can be used in conjunction with task restart to avoid restarting long-running tasks from the beginning.

Algorithmic-Based Fault Tolerance

Algorithmic-Based Fault Tolerance (ABFT) was first introduced by Huang and Abraham [1984]. They designed methods for detecting and correcting soft errors in matrix operations by adding checksum information into matrix data. The same idea has been used later on for tolerating fail-stop failures in distributed matrix computations [Chen and Dongarra, 2008; Hakkarinen and Chen, 2010; Davies et al., 2011; Du et al., 2012]. Such algorithmic recovery methods often outperform generic fault tolerance techniques such as checkpoint/restart [Graham et al., 2012]. More naturally fault tolerant algorithms can be found in the approximate computing domain. For example, monte-carlo methods generate approximate answers using a huge number of random samples. A process failure that causes the loss of some of these samples can be tolerated by either ignoring the lost samples and producing a less accurate result or replacing lost samples with new ones. Checkpointing or replication would be unnecessarily expensive for these algorithms.

2.2.3 Performance

An ideal resilient runtime system is described by Meneses et al. [2014] as "one that keeps the same execution speed despite the failures in the system". In the performance part of the taxonomy, we describe common mechanisms for minimizing the impact of failures on the overall system performance.

2.2.3.1 Performance Recovery

This property describes common mechanisms applied in distributed runtime systems for delivering sustainable performance in the presence of process failures. It only applies to runtime systems that support on-line recovery in which a failure does not shutdown the system. Hence, recovering an application does not require a manual restart.

Shrinking and Non-Shrinking Recovery

The failure of a process demands shifting the work of this process to one or more other processes. Non-shrinking recovery avoids overloading the remaining processes with additional tasks by adding new processes to compensate for the failed ones. On the other hand, shrinking recovery continues the computation using the remaining processes. Non-shrinking recovery is a highly favorable mechanism for statically balanced applications, not only for achieving better performance, but also because it relieves the programmer from the complexities of distributing the workload over an arbitrary number of processes [Laguna et al., 2014]. However, allocating new resources at runtime may be costly and even not required for certain applications. A master-worker application, for example, can use shrinking recovery and rely on its intrinsic dynamic load balancing property to achieve sustainable performance in the presence of failures, without suffering the overhead of dynamic resource allocation.

Global Load Balancing

An unbalanced distribution of the workload has negative implications on performance and resource utilization. For example, a process performing double the work performed by its peers can double the execution time of the computation while causing most of the resources to remain idle. Overloaded processes not only take longer time to compute but can also take longer time to checkpoint their state. While load balancing is essential for achieving good performance, it comes with performance overhead when performed dynamically. For that reason, multiple research efforts have targeted the design of efficient load balancing techniques [Saraswat et al., 2011; Lifflander et al., 2012; Fohry et al., 2017].

In this property of our taxonomy, we describe the programming models that provide transparent support for global load balancing — that is done by automatically migrating the work units from overloaded processes to underloaded processes.

Speculative Execution

Speculative execution avoids blocking future tasks whose preconditions are yet to be satisfied by executing these tasks based on predicted values for their preconditions. Correct predictions result in accelerating the performance, and wrong predictions result in wasted computations. Speculative execution can be used for speeding up failure recovery by launching the recovery tasks prior to the actual occurrence of the failure. For example, some big-data systems, such as Hadoop and Spark, are able to detect straggler tasks that execute at a lower speed than expected and take them as an indication for an imminent failure of the used nodes. When a straggler task is detected, the runtime system restarts the task on a different node and keeps the original task and the duplicated task work together to complete the same operation. Once one of the tasks completes, the other is killed.

2.3 Resilience Support in Distributed Programming Models

In this section, we review multiple parallel programming models and their resilience support. We start with two programming models that are based mainly on coarse-grained parallelism:

- the message-passing model (Section 2.3.1) and
- the Partitioned Global Address Space (PGAS) model (Section2.3.2)

Then, we move to a programming model that exposes both coarse-grained and fine-grained parallelism:

- the Asynchronous Partitioned Global Address Space (APGAS) model (Section 2.3.3).

Then, we move to programming models that are based mainly on fine-grained parallelism:

- the actor model (Section 2.3.4) and
- the dataflow model (Section 2.3.5)

Table 2.1 summarizes the characteristics of the different implementations of the above models according to our taxonomy.

2.3.1 Message-Passing Model

The message-passing model is a widely-used programming model for high performance computing. It is based on the Single Program Multiple Data (SPMD) model, which offers coarse-grain parallelism by executing multiple copies of the same program, each having a separate process and address space. Communication between the processes for data sharing and synchronization is performed by message passing. The standardization efforts that started in the early 1990s resulted in the MPI standard, which now has a unique portability advantage that can hardly be found in other runtime systems. The generality of the model has been proven by the diverse styles of algorithms that have been implemented using it for more than two decades. The two main open-source implementations of MPI are OpenMPI [OpenMPI, 2018] and MPICH [MPICH, 2018].

A communicator in MPI refers to a specific group of processes that are allowed to communicate with each other. A collective operation is a global communication function that involves all the processes in the communicator.

A core task for an MPI programmer is distributing the application data across the processes. Therefore, we classify MPI under the explicit resource mapping category in our taxonomy. The first release of MPI, MPI-1, provides an application with a fixed set of processes that cannot be extended during execution, and it supports only two-sided communication in which a send request at the source process is expected to be matched with a receive request at the destination process. Starting

from MPI-2, new functions have been added to support dynamic process creation and one-sided communication. More advanced features have been added to MPI-3 including support for non-blocking collectives, neighborhood collectives, and an improved one-sided communication interface.

Unfortunately, the MPI standard still lacks support for process-level fault tolerance. By default, all errors are fatal, and they leave the runtime system in an undetermined state. The lack of fault tolerance in MPI has captured the interest of many research groups, and different methods have been proposed to support system-level and user-level fault tolerance. Despite the simplicity advantage of providing fault tolerance transparently without application changes, this approach has clear limitations. First, it forces one fault tolerance technique on all applications, and second, it prevents the use of algorithmic-based fault tolerance techniques. Meanwhile, the MPI programming model is well-suited for user-level fault tolerance. That is because of adopting an explicit resource mapping policy that makes reasoning about the lost parts of the computation easy for the programmer.

In the following, we start by reviewing four representative frameworks that provide system-level fault tolerance for MPI applications: MPICH-V, FMI, rMPI, and RedMPI. After that, we review three approaches for extending MPI with user-level fault tolerance: FT-MPI, MPI-ULFM, and FA-MPI.

2.3.1.1 MPICH-V

MPICH-V is a research framework designed for studying different protocols for supporting fault tolerance in MPI without requiring any application changes [Bosilca et al., 2002]. It has been used for comparing coordinated and non-coordinated checkpointing [Lemarinier et al., 2004], as well as comparing blocking and non-blocking implementations of coordinated checkpointing [Coti et al., 2006]. Process failure is detected by a heartbeat mechanism at the TCP communication layer, which can be configured by special `keep-alive` parameters. Because MPICH-V is based on MPI-2, dynamic process creation is supported; however, the application is not required to use this capability for fault tolerance. While checkpointing is done automatically, recovering the application requires a manual restart.

2.3.1.2 FMI

The Fault Tolerant Messaging Interface (FMI) [Sato et al., 2014] is a research prototype implementation of MPI that uses in-memory checkpointing for recovering failed processes. It relies on the failure detection capability of the Infiniband Verbs Application Programming Interface (API), ibverbs, for raising communication errors at the processes directly connected to the failed process. To propagate the failure signal to the rest of the processes, FMI connects the processes in a so-called log-ring overlay network that can propagate the signal in $O(\log(n))$ messages, where n is the number of processes. To relieve the programmer from the complexities of shrinking recovery and to avoid degrading the performance after losing some resources, FMI supports

non-shrinking recovery by replacing a failed process with a spare process allocated in advance. Shrinking recovery is not supported by FMI.

2.3.1.3 rMPI

Process replication was attempted for MPI for its potential to achieve better forward progress than checkpoint/restart for highly unreliable systems and also for its ability to tolerate both fail-stop and soft errors. rMPI [Ferreira et al., 2011] uses replication to target fail-stop process failure. It implements a replication framework that creates two replicas of each MPI rank, ensures the sequential consistency of the replicas, and performs forward recovery by restarting failed replicas on pre-allocated spare nodes using the state of the corresponding active replicas. rMPI assumes the availability of a RAS service that notifies the active ranks when other ranks fail. Ropars et al. [2015] propose a user-level replication scheme for MPI that aims to achieve more than 50% resource utilization by allowing the replicas to share work rather than performing the same work twice. Application programming interfaces are provided to enable the programmer to define segments of the code that are eligible for work sharing.

2.3.1.4 RedMPI

RedMPI [Fiala et al., 2012] uses replication to detect and even correct soft faults corrupting the MPI messages. Each send request at the application level is transparently forwarded to all the replicas of the receiver. The receiver compares the received messages to detect data corruption. If three or more replicas per process are available, voting is applied to discover and drop the corrupted message. Otherwise, RedMPI terminates the application once a soft fault is detected.

2.3.1.5 FT-MPI

FT-MPI [Fagg and Dongarra, 2000] was the first significant step towards providing failure awareness to MPI applications. The application discovers the failure from the return code of the MPI function calls. Communications with the failed process will always return an error. Communications with non-failed processes can be configured to either succeed or fail based on the application requirements. When a communicator detects a communication failure, it immediately propagates the failure information to all the processes of the communicator. A communicator can be recovered by creating a new communicator using existing MPI communicator creation functions whose semantics were modified to allow the following three recovery modes: `SHRINK` — removes the failed processes and updates the ranks of the remaining processes, `BLANK` — replaces the failed processes with `MPI_PROC_NULL`, and `REBUILD` — replaces the failed processes with new ones. Collective functions were significantly modified to consider the above three modes and to ensure that a failure will not result in producing an incorrect result at the surviving processes. By the above semantic modifications, FT-MPI aims to provide enough flexibility to applications in defining the appropriate method for recovery. However, the practicality of FT-MPI has been

criticized due to introducing significant semantic changes to MPI's functions which leads to difficult library composition [Gropp and Lusk, 2004; Bland et al., 2012b].

2.3.1.6 MPI-ULFM

Driven by the increasing demand for a standard fault tolerant specification for MPI, the MPI Fault Tolerance Working Group (FTWG) was formed to meet this objective. The FTWG studied two proposals: Run Through Stabilization (RTS) [Hursey et al., 2011] and MPI User Level Failure Mitigation (MPI-ULFM) [Bland et al., 2012b]. In both proposals, failure reporting is done on a per-operation basis using special error codes as in FT-MPI. Unlike FT-MPI, both proposals avoid propagating the failures implicitly and do not guarantee uniform failure reporting in collective operations. The RTS proposal has been rejected by the MPI Forum citing implementation complexities imposed by resuming communications on failed communicators [Bland et al., 2012b]. Therefore, MPI-ULFM is currently the only active fault tolerance proposal for MPI-4.

MPI-ULFM specifies the behavior of an MPI runtime in cases of process failure and adds a minimal set of functions to provide failure propagation, process agreement, and communicator recovery services. The specification covers all MPI functions (blocking/non-blocking, point-to-point/collectives, one-sided/two-sided); however, support for one-sided communication is still lacking in the currently released reference implementation ULFM-2. Failure detection in the reference implementation of MPI-ULFM is based on heartbeating and communication timeout errors as detailed in [Bosilca et al., 2016]. The new communicator recovery operation `MPI_COMM_SHRINK` is added to facilitate shrinking recovery. By combining this operation with the existing standard function `MPI_COMM_SPAWN`, applications can implement non-shrinking recovery using dynamically created processes.

2.3.1.7 FA-MPI

Outside of the MPI FTWG, FA-MPI (Fault-Aware MPI) [Hassani et al., 2015] proposes the addition of a transaction concept to MPI to serve as a configurable unit for failure detection and recovery. Unlike MPI-ULFM, which provides failure reporting at the granularity of a single function call, FA-MPI enables the customization of the granularity of failure reporting to include one or more communication functions. FA-MPI restricts the communication within a transaction to non-blocking communications. The transaction waits for the completion of any pending communication and aggregates any detected failures. Special functions are provided to enable the application to query for the detected failures. FA-MPI allows the application to raise algorithmic-specific errors that will also be detected by the transaction, therefore it can be used for handling both soft faults and hard faults. Like MPI-ULFM, performance recovery is the application's responsibility. Both shrinking and non-shrinking recovery functions are provided to support different application requirements. The FA-MPI prototype implementation is limited to detecting failures due to segmentation faults. External daemon processes detect these failures and propagate them to the remaining processes.

2.3.2 The Partitioned Global Address Space Model

Message-passing is an effective programming paradigm for exploiting locality which is necessary for achieving scalability and good performance. However, it lacks the simplicity of direct memory access provided by shared-memory programming models. The Partitioned Global Address Space (PGAS) model emerged from interest in designing locality-aware shared memory programming models [Coarfa et al., 2005]. To achieve this goal, the PGAS model provides a SPMD model augmented with a global address space that enables inter-process communication without explicit message passing. The global address space is explicitly partitioned into local segments each owned by a process, which gives the programmer the same control over the locality of the data and its associated computation as given by MPI.

2.3.2.1 GASNet

Implementations of the PGAS model rely heavily on low-level networking libraries that provide efficient and portable support for one-sided communication and active messages, such as Global Address Space Networking (GASNet) [GASNet, 2018], the Aggregate Remote Memory Copy Interface (ARMCI) [Nieplocha et al., 2006], and ComEx [Daily et al., 2014]. The increasing criticality of fault tolerance in HPC programming models has attracted multiple research efforts to extend these libraries with fault tolerance support.

GASNet [GASNet, 2018] is widely used in many PGAS programming models. Despite its popularity, GASNet still lacks support for resilience against fail-stop process failures. GASNet-Ex is an ongoing project that aims to deliver performance and fault tolerance enhancements to GASNet. A first step towards fault tolerance has involved integrating GASNet with the Berkeley Lab Checkpoint/Restart (BLCR) system to enable transparent checkpointing for PGAS applications. Dynamic process creation and support for user-level fault tolerance are among the planned features for GASNet in the context of the GASNet-Ex project.

2.3.2.2 UPC

Unified Parallel C (UPC) [El-Ghazawi and Smith, 2006] implements the PGAS model as an extension to ANSI C. A UPC program is organized around a fixed number of execution units (threads or processes) that perform computations on shared data structures. UPC extends the C language with a `shared` pointer type for declaring global data that can be accessed by any of the execution units. Assignment statements involving shared data implicitly invoke one-sided get and put operations to copy the needed data items from their home locations. Based on shared pointers, global arrays can be allocated and processed in parallel using the `upc_forall` construct. Aggregate data transfers of shared arrays can be performed using `upc_memcpy` to achieve better communication performance. A range of constructs is provided for synchronization including locks, blocking and non-blocking barriers, and memory fences for handling data dependences. UPC supports collective data operations

such as `broadcast` and `allreduce`. However, it lacks support for multidimensional arrays and dynamic task parallelism. Zheng et al. [2014] address these limitation by designing the UPC++ library, which provides the basic UPC features in addition to supporting multidimensional arrays and dynamic task parallelism. The task model is based on a restricted form of the async-finish task model of the X10 language. Other methods for supporting dynamic task parallelism in UPC have been proposed in [Shet et al., 2009; Min et al., 2011]. UPC does not provide any resilience support against process failures.

2.3.2.3 Fortran Coarrays

Coarrays started as an extension to Fortran to support parallel processing by applying the PGAS model [Numrich and Reid, 1998; Numrich, 2018]. It is now a core feature of the Fortran 2008 and the Fortran 2018 standards. This model enables parallel execution using a fixed number of images (i.e. processes) that can share data in the form of *coarrays*. A coarray is a global variable partitioned among images. Unlike UPC, which provides uniform abstractions for accessing local and remote date, Coarrays makes partitioning explicit in the program. To access a particular coarray item, the dimensions of the item and the image number must be specified in the program.

The Fortran 2018 standard added the concept of *failed images* as a means for supporting user-level fault tolerance. Programmers detect image failures by inspecting error codes returned from coarray operations. The new interface `failed_images` was added to assist programmers in identifying the failed images. Fanfarillo et al. [2019] added these features to OpenCoarrays [OpenCoarrays, 2019] using MPI-ULFM.

2.3.2.4 GASPI

The Global Address Space Programming Interface (GASPI) [Simmendinger et al., 2015] is a fault-tolerant library implementation of the PGAS model that is supported for C, C++ and Python and can interoperate with MPI. GASPI aims to enable efficient use of one-sided communication in high-level programming models by utilizing the available multi-core parallelism in modern architectures. The program can register read/write operations and notification actions in multiple queues that can be processed in parallel by the runtime system. The notification actions are used for synchronization and failure detection. Invoking `gaspi_wait` on a notification action blocks the caller until all read/write operations started before the notification complete. Each read/write operation is registered with a time-out value. Operations that do not complete within the specified time limit indicate the possibility of a process failure. The program can validate the status of a specific process using the `gaspi_state_healthy` function. Failure knowledge is expected to be propagated implicitly to all the processes, to enable the program to switch to a global recovery phase. Bartsch et al. [2017] use GASPI failure notification to support online non-shrinking recovery for PGAS applications by applying in-memory coordinated checkpointing.

2.3.2.5 OpenSHMEM

OpenSHMEM [OpenSHMEM, 2017] is an effort to provide a standard programming interface for implementing the PGAS model. The execution model is based on a group of processing elements (PEs), each having a private memory space for allocating non-shared variables, and a so-called symmetric heap for allocating shared variables that can be accessed by any PE. Allocating a shared variable is performed by allocating an image of the variable in the symmetric heap of each PE. A variety of interfaces are provided for memory management, synchronization, one-sided put/get operations, and atomic operations. Different research groups have proposed fault tolerance extensions for OpenSHMEM. Bouteiller et al. [2016] propose extensions to OpenSHMEM to support the same user-level fault tolerance mechanism applied in MPI-ULFM. Garg et al. [2016] use the Distributed MultiThreaded CheckPointing (DMTCP) [Ansel et al., 2009] system to provide transparent system-level checkpointing with offline recovery. Hao et al. [2014a] support coordinated in-memory checkpointing by adding the new collective operation `shmem_checkpoint_all`, which stores a replica of each symmetric heap at the neighboring PE and raises an error if any of the PEs is dead. The program can respond to a reported failure by shrinking the application to the remaining PEs or create new PEs dynamically using the proposed `shmem_restart_pes` operation.

2.3.2.6 Global Arrays

In addition to general-purpose programming languages, the PGAS model has been used in libraries of distributed data structures. Global Arrays (GA) [Nieplocha et al., 1996] extends MPI with shared array abstractions. Fault tolerance for GA applications has been explored using checkpoint/restart [Tipparaju et al., 2008], matrix encoding techniques [Ali et al., 2011b], and data replication [Ali et al., 2011a].

2.3.2.7 Global View Resilience

Global View Resilience (GVR) [Chien et al., 2015] is another library that provides the same global array functionalities of the GA library. Moreover, it provides data versioning API that supports a wider spectrum of recovery options compared to coordinated single-version checkpointing. Users control when to create new versions and how to use the available versions to recover the application state after a failure, thus it provides a flexible foundation for ABFT.

2.3.3 The Asynchronous Partitioned Global Address Space Model

The coarse-grain parallelism model provided by MPI and PGAS, although suited for many regular HPC applications, is highly limited in exploiting the available fine-grain parallelism in mainstream architectures. The Asynchronous Partitioned Global Address Space (APGAS) model addresses this limitation by extending the PGAS model with general task parallelism. Each task has an affinity to a particular partition; however, it can spawn tasks at all other partitions. In the following, we describe two widely-known APGAS languages: Chapel and X10. Both languages

initially emerged as part of DARPA's High Productivity Computing Systems (HPCS) project, which aimed to advance the performance, programmability, portability and robustness of high-end computing systems.

2.3.3.1 Chapel

Chapel [Chamberlain et al., 2007] is an object-oriented APGAS language developed by Cray Inc. It provides a uniform programming model for programming shared-memory systems and distributed-memory systems. For the latter, it supports a number of communication libraries, such as GASNet and Cray's user Generic Network Interface (uGNI), for performing data and active-message transfer operations.

Chapel's execution model is organized around a group of multithreaded *Locales*, on which tasks and global data structures can be created. It provides the *domain* data type for describing the index set of an array, which can be dense, sparse, or in any other user-defined format. A *domain map* describes how the domain is mapped to a group of locales. Assignment statements involving remote data translate implicitly to one-sided get/put operations. For locality control, the `on` construct is provided to determine the locale on which a particular active message should execute. In addition to locality control and global-view data-parallelism, Chapel provides simple abstractions for expressing task-parallelism.

Task parallelism is supported by the following constructs, which can be composed flexibility to express nested parallelism:

- `begin`: spawns a new task that can execute in parallel with the current task.
- `cobegin`: spawns a group of tasks, one for each statement in the cobegin scope, and waits for their termination.
- `(co)forall`: spawns a group of parallel tasks for processing the iterations of a loop, and waits for their termination.
- `sync` statement: waits for all dynamically created begins within its scope.

In addition to the `sync` statement described above, Chapel provides a `sync` variable type that can also be used for synchronization. A `sync` variable has a value and a state that can be either *full* or *empty*. Reading or writing a `sync` variable can cause the enclosing task to block depending on the state of the variable [Hayashi et al., 2017]. For example, reading an empty variable or writing to a full variable will block execution until the state changes to the opposite state. Coordinating the different tasks can therefore be achieved by orchestrating their access to shared `sync` variables.

Currently, Chapel is not resilient to process failures. A recent research effort by Panagiotopoulou and Loidl [2015] targeted the challenges of transparently recovering the control flow of a Chapel program when locales fail. A copy of each remote task is replicated at the parent locale that spawned the task. When a locale fails, other locales identify the lost tasks and re-execute them locally. The proposed design is limited to tasks that have no side effects. Their future work plans include handling

general tasks that alter the data. GASNet was used as the underlying communication layer for Chapel. Because GASNet is not resilient, this study simulated the existence of failures rather than actually killing locales.

2.3.3.2 X10

X10 [Charles et al., 2005] is an object-oriented APGAS language developed by IBM. The language syntax is based on the Java language, with the addition of new constructs for expressing task parallelism and global data. Two runtime implementations are available for X10: *native* X10 (executes X10 programs translated into C++ code) and *managed* X10 (executes X10 programs translated into Java code).

A *place* is the locality unit in X10. The program starts from a root task at the first place and evolves by dynamically creating more tasks at the different places. The programmer specifies the place where a certain task should execute using the `at` construct. X10 supports nested task parallelism using the async-finish model. The `async` construct is similar to Chapel's `begin` construct; it creates an asynchronous task at the current place. The `finish` construct is similar to Chapel's `sync` statement; it waits for the termination of all asyncs spawned dynamically within its scope. Using `async`, `finish`, and `at`, the programmer can express not only simple fork-join task graphs but also more complex task graphs with arbitrary synchronization patterns.

For handling global data, X10 provides a *global reference* type that carries a globally unique address for an object; it is similar to the `shared` pointer primitive of UPC. However, unlike all PGAS models we described above in which a global reference can be used for transferring data implicitly between processes, X10's global references do not permit any implicit communication. To access an object using its global reference, a task must be created at the home place of the object where the object will be accessed locally. Otherwise, bulk-transfer functions are provided in the standard library to transfer arrays using their global references. The runtime system can implement these functions via one-sided put/get operations or normal two-sided communications (possibly at a higher performance cost). In all cases, data transfer is explicit, and the cost of communication is obvious in the program.

Recently, the X10 team at IBM has focused on improving the language's resilience to process fail-stop failures [Cunningham et al., 2014]. They adopted a flexible user-level fault tolerance approach that enables the development of generic as well as algorithmic-based fault tolerance techniques at the application level. Native X10, which is designed for HPC, is currently limited to fixed resource allocation. However, spare places can be allocated in advance for supporting non-shrinking recovery.

Hao et al. [2014b] describe X10-FT, an extension of X10 that supports transparent disk-based checkpointing. The compiler is modified to insert checkpointing instructions at program synchronization points. The places are organized hierarchically; when a place fails, its parent place creates a replacement place and initializes it using the latest disk checkpoint. The PAXOS protocol is used for failure detection (via heartbeating) and for reaching consensus on the status of places.

The X10 programming model will be described more fully in Section 2.4.

2.3.4 The Actor Model

The actor model represents a system as a group of concurrent objects, named actors, each owns a private state and an inbox to receive asynchronous messages from other objects. Based on the content of a received message, an actor may update its own state, send messages to other actors, or create new actors. The actor model encourages work over-decomposition to fine-grain components, which is an essential property for harnessing the massive parallelism available in current many-core systems. Location transparency is a common feature in many actor programming models, where mapping the actors to the underlying system resources is done implicitly by the runtime system.

The actor model is an attractive model for fault tolerance. The strong encapsulation of data and behavior makes an actor an autonomous entity that may be recovered independently, depending on the level of coupling with other actions. Different implementations of the actor model provide different levels of failure awareness to applications. Some models recover lost actors transparently without any user intervention, while others provide application-level mechanisms for failure detection and recovery.

In the following, we describe Charm++, a well-known implementation of the actor model in the HPC domain. Followed by a review of three famous non-HPC actor implementations: Erlang, Akka and Orleans.

2.3.4.1 Charm++

Charm++ [Kale and Krishnan, 1993; Acun et al., 2014] is the most prominent implementation of the actor model in HPC. It defines a computation as a collection of migratable actors named chares. The runtime system transparently manages the distribution of chares among the processes, and it may migrate the chares for load balancing or for avoiding imminent faults. Charm++ applies a fixed resource allocation policy; however, the applications can allocate spare processes to be used for failure recovery. The highly adaptive execution model of the Charm++ runtime system enables it to handle failure recovery without user intervention. Recovery from process-level hard faults has been explored using blocking [Zheng et al., 2004] and non-blocking [Ni et al., 2012] coordinated checkpointing, and using uncoordinated checkpointing via pessimistic message logging [Chakravorty and Kale, 2004] and causal message logging [Meneses et al., 2011]. Recovering from both hard faults and soft faults by combining checkpointing and replication has been supported in the Automatic Checkpoint/Restart (ACR) system [Ni et al., 2013] based on Charm++. Because Charm++ masks process failures from the application, it is not a suitable target for algorithmic-based fault tolerance. It has also been criticized for complexities in expressing global control flows and global communications which are common in many HPC applications [Jain et al., 2015].

Adaptive MPI (AMPI) [Huang et al., 2003] is a fault tolerant implementation of the MPI programming model on top of Charm++. The MPI processes are implemented as light-weight user-level threads mapped to Charm++ chares. Message passing between

the MPI processes is realized by the message-driven communication of Charm++. By checkpointing the chares to disk, AMPI applications can be restarted after failure using the same or a different number of nodes. Online recovery using proactive migration has also been supported in AMPI [Chakravorty et al., 2006].

2.3.4.2 Erlang

Erlang [Vinoski, 2007] emerged originally from the need to build highly reliable telecommunication systems. Erlang actors are mapped explicitly to the available nodes and cannot migrate automatically for load balancing. An interesting feature in Erlang is the organization of actors in a supervision tree that facilitates recursive fault handling. A failing actor results in generating an error message at its parent who can perform certain recovery actions — for example, restart the failed actor on another node. The root actor is the main guardian that handles failures that escalate to the top of the tree. Therefore, while Charm++ programs are oblivious to failures, Erlang programs are aware of failures because failure handling is defined at the user level.

2.3.4.3 Akka

Akka [Akka, 2018] is a library implementation of the actor model for Scala and Java programs. The programming model provides location transparency for actors (similar to Charm++) and recursive failure handling using supervision trees (similar to Erlang). Akka supports elasticity by allowing the compute nodes to join and leave the cluster flexibly during execution. The status of nodes is monitored via a heartbeating mechanism. Actors that were hosted at a failed node can be recreated by their supervisors, possibly using a previously checkpointed state.

2.3.4.4 Orleans

Orleans [Bykov et al., 2011] is an actor-based elastic framework for cloud computing. Aiming to provide a higher-level programming model than Erlang and Akka, Orleans takes full control over creating, restoring, and deleting actors. It supports location transparency and integrates multiple mechanisms for handling process failures [Bernstein et al., 2014]. It uses in-memory replication for improving the system's availability, disk-based checkpointing for restoring lost actors, and resilient transactions for handling atomic actions on multiple actors.

2.3.5 The Dataflow Programming Model

The dataflow programming model expresses a computation as a graph of nodes representing tasks and edges representing dependencies between the tasks. Similar to the actor model, the dataflow model encourages work over-decomposition and decoupling the work units (tasks) from the processes that execute them. However, there are two major differences between the two models. First, in the actor model, tasks and their data are strongly coupled and always co-located, while in the dataflow model, tasks and data are decoupled which makes locality optimizations more challenging.

Second, in the actor model, the programmer implicitly encodes the dependencies using synchronization instructions that enforce scheduling constraints on the actions, while in the dataflow model, the programmer explicitly defines the task dependencies to the runtime system [Aiken et al., 2014]. Making the runtime system aware of the dependencies enables it to automatically handle task scheduling, synchronization, and overlapping communication and computation. In the following, we review different distributed dataflow systems and describe their resilience features.

2.3.5.1 OCR

The Open Community Runtime (OCR) [Mattson et al., 2016] is a dataflow runtime system designed as a low-level target for high-level task-based programming models. The main concepts of the OCR programming model are: event-driven tasks (EDTs), events, datablocks, and hints. EDTs are non-blocking tasks that may have dependencies on certain datablocks and/or other tasks. A datablock is the storage unit of the application. The runtime manages the placement, access control, and replication decisions of datablocks. The location and number of copies of a datablock can change during the runtime for locality handling, failure avoidance, or load balancing. Events are used to represent control and dependencies between tasks. Although task scheduling and data placement decisions can be handled automatically, OCR programmers can provide performance tuning hints to influence these decisions. OCR is well-suited for supporting resilience; however, current implementations are not resilient to soft faults or hard faults.

2.3.5.2 Legion

The Legion [Bauer et al., 2012] programming model makes locality control central to its dataflow model by organizing the computation around tasks and logical regions. The logical region abstraction enables the program to express a hierarchical view of the application data by partitioning them into regions and sub-regions. The regions can be disjoint or overlapping and can be accessed with a particular privilege by different tasks (i.e. read-only, read-write, or reduce). Legion's distributed work-stealing scheduler takes into account the locality information of the regions, the task dependencies, and the access privileges to extract hidden parallelism in the program. To extract even more parallelism, Legion supports speculative execution for predicated tasks whose execution depends on a special boolean parameter (for example, a parameter representing the convergence status of an algorithm). Based on the runtime's prediction of this boolean parameter, a task can start execution before the parameter value is evaluated. Mispredictions are handled by ignoring the results of the speculated tasks and rolling back any changes they applied on the regions. Bauer [2014] outlines a transparent mechanism for recovering from soft faults and hard faults by restarting failed tasks. Failure detection is assumed to be done by an external system. However, no performance results are reported for Legion applications under failures. Because Legion is based on the non-resilient

communication library, GASNet, we do not expect it to survive any process failures in practice.

2.3.5.3 NABBIT and PaRSEC

NABBIT [Kurt et al., 2014] and PaRSEC [Cao et al., 2015] are dynamic task-based runtime systems. Both support resilience against soft errors. NABBIT uses task re-execution without checkpointing for recovery. On the other hand, PaRSEC supports three recovery models: task re-execution, task re-execution with checkpointing, and ABFT. NABBIT implements a distributed work-stealing scheduler, whereas PaRSEC lacks this feature. Tolerating fail-stop process failure has not been supported yet in either system.

Other examples for task-based dataflow systems for HPC include HPX [Kaiser et al., 2014], StarPU [Augonnet et al., 2011], and Realm [Aiken et al., 2014]. However, none of them is resilient to failures.

2.3.5.4 Spark

From the cloud computing domain, Spark [Zaharia et al., 2012] is a widely-used distributed dataflow system optimized for iterative processing and interactive data mining. The execution of a Spark job starts at a driver node, which transforms the job into a directed acyclic graph and submits its tasks to the available worker nodes. Spark offers a data abstraction called Resilient Distributed Datasets (RDD), a form of immutable distributed collections that are cached in memory. When a worker node fails, an RDD may lose some of its partitions. In order to restore lost partitions, Spark tracks the RDD lineage, which is the history of operations that were performed to create an RDD. The driver node replays the lineage operations to recover lost RDD partitions. For applications with long computation lineage, user-controlled checkpointing can be applied to speed up recovery using the last checkpoint rather than recomputing the RDD from the initial application state. Spark applies speculative execution by monitoring the progress of tasks and resubmitting the straggler tasks on other nodes.

2.3.6 Review Conclusions

Table 2.1 summarizes the resilience characteristics of distributed processing systems discussed in this section. We make the following observations:

- user-level recovery of process hard faults is gaining increasing adoption in MPI and PGAS, but is absent in the dataflow model.
- limited research is targeting soft faults in parallel programming models.
- non-shrinking recovery is the most adopted method for performance recovery in SPMD programming models, and global load balancing is the most adopted method in the actor and dataflow model. Speculative execution is more popular in big-data systems, such as Hadoop and Spark, than in HPC systems.

Table 2.1: Distributed processing systems resilience characteristics.

Distributed System	Adaptability		Fault Tolerance					Performance
	Resource Allocation d=dynamic f=fixed f*=fixed (+spare)	Resource Mapping i=implicit e=explicit	Fault Type h=hard h*=hard (design only) s=soft	Fault Level t=task p=process	Recovery Level u=user s=system	Fault Detection hb=heartbeat ce=comm/err ex=external p=prediction di=data inspect.	FT Technique c/r=ckpt/restart rep=replication mig=migration tr=task/restart tx=transaction ABFT=ABFT	Performance Recovery sh=shrinking nsh=non-shrinking glb=global load balancing spec=speculative exec.
MPI								
MPI-1	f	e	✗	✗	✗	✗	✗	✗
MPI-2/3	d	e	✗	✗	✗	✗	✗	✗
MPICH-V	d	e	h	p	s	hb	c/r	✗
FMI	f*	e	h	p	s	ce	c/r	nsh
rMPI	f	e	h	p	s	ex	rep	nsh
RedMPI	f	e	s	p	s	di	rep	✗
AMPI	f	e	h	p	s	ce, p	c/r, mig	glb
FT-MPI	d	e	h	p	u	ce	ABFT	sh/nsh
MPI-ULFM	d	e	h	p	u	hb/ce	ABFT	sh/nsh
FA-MPI	d	e	h/s	t/p	u	ex	tx	sh/nsh
PGAS								
UPC	f	e	✗	✗	✗	✗	✗	✗
F2008 Coarrays	f	e	✗	✗	✗	✗	✗	✗
F2018 Coarrays	f	e	h	p	u	ce	ABFT	✗
GASPI	d	e	h	p	u	ce	ABFT	sh/nsh
GASPI (C/R)	f*	e	h	p	s	ce	c/r	nsh
OpenSHMEM	d	e	h	p	u/s	ce	ABFT, c/r	sh/nsh
APGAS								
Chapel	f	e	✗	✗	✗	✗	✗	✗
Chapel (prototype)	f	e	h*	p	s	ce/ex	rep	✗
X10	f	e	h	p	u	ce	ABFT	✗
X10-FT	d	e	h*	p	s	hb	c/r	nsh
Actor								
Charm++	f	i	h	p	s	ce	c/r	glb
Charm++ ACR	f*	i	h/s	p	s	ce & di	c/r & rep	glb/nsh
Erlang	d	e	h	p	u	ex	ABFT	sh/nsh
Akka	d	i	h	p	u	hb	ABFT, c/r	glb
Orleans	d	i	h	p	s	hb	c/r, tx, rep	glb
Dataflow Systems								
OCR	f	i/e	✗	✗	✗	✗	✗	✗
Legion	f	i/e	h*	t/p	s	ex	tr	glb, spec
PaRSEC	f	i/e	s	t	u/s	di	ABFT, c/r, tr	✗
NABBIT	f	i	s	t	s	di	tr	glb
Spark	d	i	h	p	s	hb	c/r, tr	spec

- checkpoint/restart is the most adopted fault tolerance technique. Checkpointing frameworks that provide online recovery rely on coordinated neighbor-based in-memory checkpointing. Examples include: FMI, GASPI, and OpenSHMEM.

- to date, PGAS and APGAS parallel programming models have not utilized transactions for achieving fault tolerance. Recent advances in supporting resilient transactions for PGAS-based distributed systems are promising for addressing this gap [Dragojević et al., 2015; Chen et al., 2016].

- failure detection is typically achieved by receiving communication errors or by heartbeating.

- there is a high demand for extending GASNet with process-level fault tolerance due to its wide adoption in many programming models, such as UPC, Legion, Chapel, and OCR.

We now consider the question: **which programming model is more suitable for supporting multi-resolution resilience in the presence of process-level hard faults?** Multi-resolution resilience is a special type of user-level resilience that gives the control over optimizing the application's resilience performance to the programmer. Handling process-level failures at the application level requires application awareness of the process's boundaries, so that the application can identify the lost work when failures occur. Process-level parallelism is an essential programming abstraction only in the message-passing, PGAS, and APGAS models[1]. Therefore, we consider these models more adequate for supporting multi-resolution resilience than the actor and dataflow models. The other two factors of multi-resolution resilience are performance and productivity. Only the APGAS model provides both locality-awareness and fine-grained parallelism, which are essential features for achieving scalability. Productivity is also a key objective for APGAS languages, which is achieved by providing a global address space and general support for task parallelism via high-level constructs. Therefore, we conclude that the APGAS model is the most suitable model to reconcile resilience, performance and productivity for multi-resolution resilience.

In this thesis, we study multi-resolution resilience in the context of the X10 language for the following reasons. The initial steps taken for supporting resilience in X10 [Cunningham et al., 2014] make it a promising target for performing a deeper study on the practicality of the proposed model, in terms of performance and productivity. The lack of fault tolerant Remote Direct Memory Access (RDMA) support in most communication libraries hinders many PGAS languages from supporting resilience. Fortunately, X10 is not bound to using RDMA operations; its active messages and data transfers can be implemented entirely using two-sided communication operations. This allows us to focus on the internals of control flow resilience and data resilience, rather than the challenges of supporting fault tolerant RDMA operations.

[1] In other words, giving users control over mapping work units to processes is a necessary feature in MPI, PGAS and APGAS, but optional in actor and dataflow.

2.4 The X10 Programming Model

X10 models a parallel computation as a directed-acyclic task graph (task-DAG) that spans a group of places. A place is an abstraction for an operating system process that contains a collection of data and tasks operating on these data. The X10 type `x10.lang.Place` represents a place, and `x10.lang.PlaceGroup` represents a collection of places. The method `Place.places()` returns a group of all the places currently available for the computation. The failure of places or the creation of new places dynamically alters this group, as we will explain in Section 2.4.5.

2.4.1 Task Parallelism

Each task has an affinity to a specific place defined by the user. A task is allowed to spawn other tasks locally or remotely and to block during execution to wait for the completion of other tasks. The following three constructs are provided by X10 for expressing task parallelism.

- The **`async`** construct: for spawning a new task.
- The **`at`** construct: for specifying task affinity.
- The **`finish`** construct: for task synchronization.

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`. Any values at the spawning place that are accessed by `S` are automatically serialized and sent to place `p`. 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 wrapped in a `MultipleExceptions` object that is thrown 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`. An error raised while executing `S` synchronously is thrown by **`at`** in an `Exception` object.

2.4.2 The Happens-Before Constraint

The **`finish`** and **`at`** constructs define synchronization points in the program that force happens-before constraints between statements at different places.

The use of **`finish`** in the following code implies that statement `S` must happen before statement `E`. Any task created transitively by `S` delays the completion of the finish statement until after the termination of the task.

```
1 finish S;
2 E;
```

The use of **at** in the following code implies that statement S at place q must happen before statement E at place p. Unlike **finish**, **at** does not wait for other asynchronous tasks spawned by S.

```
1 //from place p
2 at (q) S;
3 E;
```

On the other hand, the **async** construct is used to express possible parallelism between statements. For example, the use of **async** in the following code implies that statement S at place q and statement E at place p can proceed in parallel.

```
1 //from place p
2 at (q) async S;
3 E;
```

Figure 2.2 shows a sample X10 program with the corresponding task DAG. The first finish block F1 tracks two asynchronous tasks a and b. Task b creates the second finish block F2 which tracks task c and its successor task d. Thanks to F2, tasks c and d must complete before the message at Line 9 is printed. Similarly, thanks to F1, the four tasks must complete before the message at Line 12 is printed. These happens-before 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');
```

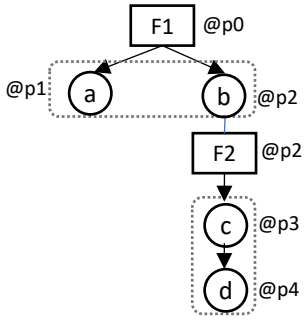

Figure 2.2: A sample X10 program and the corresponding task graph. A dotted box represents a finish scope.

2.4.3 Global Data

Each object has a fixed affinity to a specific place and can only be manipulated by tasks running at the same place. A place can expose global references to its objects to enable global data access. Two types are available for that purpose: `x10.lang.GlobalRef` and `x10.lang.PlaceLocalHandle`. The first creates a global reference for a single object. The second creates a global reference for a family of objects, one per place for a certain place group.

X10 requires users to explicitly handle data movement, thereby making the cost of remote access obvious in the program. Depending on the object size and other possible criteria, a choice can be made between transferring a deep copy of the object or only sending its global reference to the destination task. A task that holds a global reference can reach the remote object indirectly by spawning backward tasks at the object's home place. The following code is an example for accessing a remote object of type `ResultContainer` using a global reference. Dereferencing a global reference or a place-local handle is done using the operator `()`, as shown in Line 7. The atomic statement at Line 6 is used to guard against possible races due to accessing the `result` object from concurrent tasks.

```
1 val result = new ResultContainer();
2 val gr = GlobalRef[Result] (result);
3 at (p) async {
4    val partial = compute_partial_result();
5    at (gr) async {
6        atomic {
7            val _result = gr();
8            _result.merge(partial);
9        }
10   }
11 }
```

2.4.4 Resilient X10

Resilient X10 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. The protocol is designed to adhere to the following principle, named the Happens-Before Invariance (HBI) principle:

> failure of a place must not alter the happens-before relation between statement instances at other places [Cunningham et al., 2014]

Let us examine the HBI principle in the context of Figure 2.2. Assume that place `p2` died after tasks `c` and `d` were created. Now tasks `c` and `d` are orphaned because their parent finish `F2` is lost. Suppose `F1` ignored these tasks and terminated. That would allow Line 12 to run in parallel with `c` and `d`, which breaks the happens-before relation that the program defined. There are two options to handle tasks `c` and `d` to keep the happens-before relation; either kill them or have their grandparent adopt them (i.e. force `F1` to wait for `c` and `d`). Because killing the tasks would leave the

heap in an undetermined state, Cunningham et al. [2014] designed their fault-tolerant termination detection protocol based on adoption.

2.4.5 Elastic X10

Elasticity is the system's ability to shrink and expand the resources it uses at run-time. It is an attractive feature for fault tolerance as it enables applications to leverge non-shrinking recovery. Elasticity support has been added recently to the X10 language by supporting dynamic place creation. New places can be created externally or requested in the program by calling `System.addPlaces(n)` or its synchronous version `System.addPlacesAndWait(n)`. Currently, dynamic place creation is only implemented in the Managed X10 runtime. Native X10 programs need to allocate spare places in advance to use non-shrinking recovery. Bungart and Fohry [2017b] addressed this limitation by adding elasticity support for native X10 programs running over MPI-ULFM. However, the implementation is not published yet.

2.4.5.1 The PlaceManager API

The typical method for identifying the group of available places in X10 is by calling `Place.places()`. It returns a sorted `PlaceGroup` indexed by the place id. Changes in the available places are reflected in `Place.places()` — failed places are removed, and dynamically created places are included at the end of the group. For example, in Figure 2.3, after the failure of place 1, place 5 was added to the end of the group. The X10 runtime does not reuse the ids of failed places for the new places. Because a newly added place has neither the same identity nor the same order, reorganizing the places manually to make them more aligned with the structure used before the failure might be unavoidable in many applications.

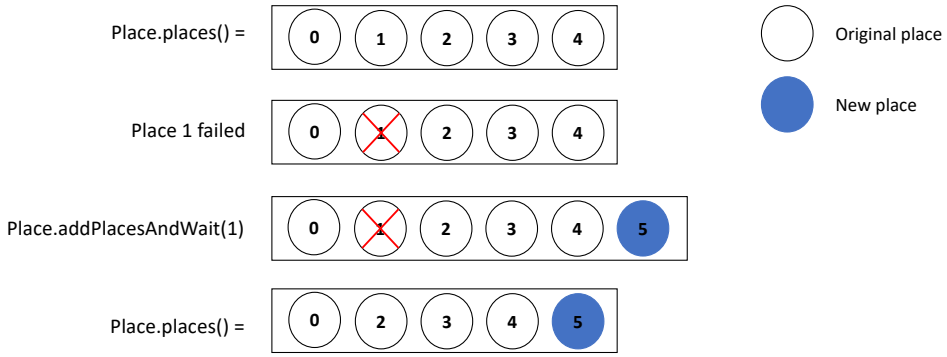

Figure 2.3: Place management with Place.places().

To simplify place management for non-shrinking recovery, the X10 team added the class `PlaceManager` to the standard library. A `PlaceManager pm` splits the available places into active places and spare places, as presented in Figure 2.4. The program is expected to distribute its work over the group of active places identified

by `pm.activePlaces()`. When an active place fails, the program rebuilds the activePlaces group by calling `pm.rebuildActivePlaces()`. This method replaces each failed place with a spare place *in the same ordinal location*. When a spare place is activated, a new spare place is created asynchronously to replace it, only if the runtime system supports dynamic place creation. The result of `pm.rebuildActivePlaces()` is a `ChangeDescription` instance that includes the list of `addedPlaces()` and the list of `removedPlaces()`. The program can use this knowledge to initilize the newly added places as a precondition for resuming the computation.

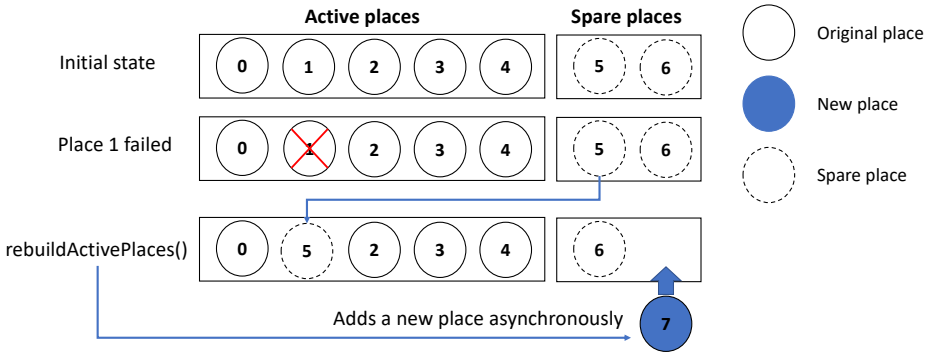

Figure 2.4: Place management with the PlaceManager class.

Note that the three methods `Place.places()`, `pm.rebuildActivePlaces()`, and `pm.activePlaces()` are all local functions. They are aimed to reflect the state of the system as viewed by the calling place. Therefore, different results may be returned at different places depending on when a certain place detects the failure or the addition of a place. However, in practice, identifying the available places is often required only by the main task at place-zero that supervises the execution of the program. The following code shows a typical example for using the PlaceManager API in a resilient X10 program.

```
1 def execute(pm:PlaceManager) {
2     var success:Boolean = false;
3     while (!success) {
4         try {
5             val pg = pm.activePlaces();
6             compute_over(pg); /*performs the computation over
7                                 the given place group*/
8             success = true;
9         } catch (e:DeadPlaceException) {
10            pm.rebuildActivePlaces();
11        }
12    }
13 }
```

2.4.5.2 Place Virtualization

Because the `PlaceManager` preserves the order of the active places throughout the execution, the index of a place in the activePlaces group can be used in the program as a *virtual identifier* for the place, rather than using the physcial place id. By this simple coding strategy, minimal changes will be required for adding resilience to the existing source code. We follow this strategy in the implementation of all resilient applications presented in the thesis.

2.5 Summary

This chapter presented a taxonomy of resilient programming models and a broad overview of resilience features in different programming models. It described the rationale behind the choice of the APGAS model for supporting multi-resolution resilience and described the details of the X10 programming model. In the following chapter, we describe how we improved the performance scalability of RX10 using MPI-ULFM — an efficient fault tolerant implementation of MPI.

Chapter 3

Improving Resilient X10 Portability and Scalability Using MPI-ULFM

This chapter describes how we enabled RX10 to scale to thousands of cores in supercomputer environments using MPI-ULFM. It details the matching semantics between MPI-ULFM and the transport layer of RX10. It also describes the methods used to deliver global failure detection for resilient finish, and to speed up the performance of X10's fault-tolerant collectives. We evaluate our implementation using microbenchmarks that demonstrate the efficiency and scalability advantages that RX10 applications can achieve by switching from the socket transport to the MPI-ULFM transport. The experiments focus on failure-free scenarios only. We evaluate the performance of applications in failure scenarios in Chapter 5.

After the introduction in Section 3.1, we describe how X10 uses MPI in non-resilient mode in Section 3.2. Then we describe the MPI-ULFM specification in Section 3.3. After that we explain how we integrated MPI-ULFM with the X10 runtime system in Section 3.4. We conclude with the performance evaluation in Section 3.5.

This chapter is based on work published in the paper "Resilient X10 over MPI user level failure mitigation" [Hamouda et al., 2016].

3.1 Introduction

MPI is the standard communication API on supercomputers. Compared to other communication APIs, such as TCP sockets or Infiniband verbs, MPI is simpler to use for writing distributed programs. For example, MPI transparently establishes the connections between processes, assigns a rank for each process, and matches a rank to its IP address and port on behalf of the programmer. In contrast, a socket API would require the programmer to manually handle these details. MPI implementations are portable over different networking hardware, and they often strive to make the best use of the available hardware capabilities to provide optimized point-to-point and collective operations. While supercomputers are dominated by high-performance computing applications following MPI's SPMD model, cloud environments have

more diverse applications with different networking requirements. Cloud applications are more likely to use lower-level communication APIs than MPI to directly establish, monitor, and destroy the connections between processes. TCP socket APIs are commonly used for inter-process communication on cloud environments. When a process fails, connections to that process are invalidated and errors are raised at the other end of the connection. Based on this failure notification capability, fault-tolerant applications and runtime systems can be built on top of sockets.

X10 can be used for developing high-performance computing applications as well as cloud-based services. It supports both MPI and TCP sockets for inter-process communication. Figure 3.1 shows the layered architecture of the X10 runtime. The middle layer, X10 Runtime Transport (X10RT), is an abstract transport layer that delegates communication requests from the runtime to one of the concrete transport implementations. The latest version of X10 at the time of writing (v2.6.1) provides three X10RT implementations: a TCP sockets implementation, an MPI implementation, and an implementation for the Parallel Active Message Interface (PAMI) — IBM's proprietary API for inter-process communication on supercomputers.

The X10 runtime executes the tasks using a pool of worker threads. The communication activities of the tasks are performed by the same worker threads. To avoid blocking the worker threads, which may lead to deadlocks, the X10 runtime requires *non-blocking communication support* from the underlying transport layer.

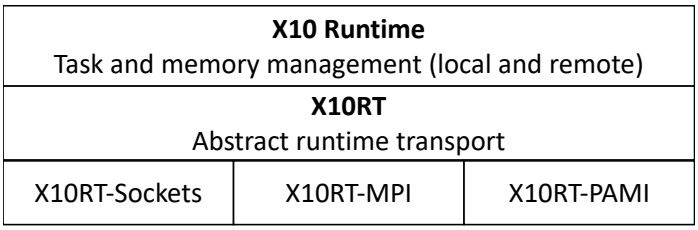

Figure 3.1: X10 layered architecture.

The initial development of RX10 was primarily concerned with bringing X10 to cloud computing environments. Therefore, it only made resilient the portions of the X10 runtime stack that were appropriate for a cloud computing environment. Thus, only X10RT-Sockets was modified to survive place failure. X10RT-Sockets opens Secure Shell (SSH) connections to launch processes at remote nodes — a mechanism that is often not permitted by supercomputers for security reasons. On the other hand, the majority of supercomputers permit launching processes using MPI. Because both MPI and PAMI are not fault-tolerant, RX10 was not portable to supercomputers. Experimental evaluations ran on small clusters and scaled to only a few hundred X10 places.

At the same time RX10 was being implemented, the MPI FTWG and members of the Innovative Computing Laboratory at University of Tennessee were actively developing MPI-ULFM [Bland et al., 2012a,b, 2013], a fault tolerance specification for MPI. The proposed specification is planned to be part of the upcoming MPI-4 [2018]

standard. The MPI-ULFM team publishes the latest updates about the project on the fault tolerance research hub (fault-tolerance.org).

A reference implementation of MPI-ULFM is available based on the open source MPI implementation OpenMPI [2018]. The first release, ULFM-1, was based on an old version of OpenMPI (v1.7) compatible with the MPI-2 standard, which does not support non-blocking collectives. ULFM-1 also did not have proper support for thread safety. ULFM-2 was released in April 2017 based on OpenMPI (v3.0), compatible with the MPI-3 standard. It provides non-blocking collectives and lightly tested support for fault tolerance with multiple threads. ULFM-2 is used as a basis for our experimental analysis in Section 3.5.

The popularity of MPI-ULFM has been increasing, and many researchers started relying on it for building resilient MPI applications. Pauli et al. [2013] used ULFM for resilient Monte Carlo simulations. Ali et al. [2014, 2015] implemented exact and approximate recovery methods for 2D and 3D Partial Differential Equation (PDE) solvers over ULFM. Laguna et al. [2014] provided a programmability evaluation for ULFM in the context of a resilient molecular dynamics application.

The development of MPI-ULFM served as a promising opportunity for us to shift RX10 to supercomputer scale. In the following sections, we detail how we integrated MPI-ULFM with X10 to provide failure notification and optimized fault-tolerant collectives at large scales. To the best of our knowledge, our work is the first to evaluate ULFM in the context of a high-level language.

3.2 X10 over MPI

In this section, we describe how X10 uses MPI with a focus on initializing X10 places, handling active messages, and handling collective operations.

3.2.1 Initialization

X10 programs can choose between sockets, MPI, or PAMI as a compilation target. When MPI is chosen, X10 places are launched as normal MPI processes. They start by initializing the default communicator `MPI_COMM_WORLD` by calling `MPI_INIT_THREAD(..)` and specifying the desired thread support. MPI provides four levels of thread support:

- `MPI_THREAD_SINGLE`: for single-threaded MPI processes.
- `MPI_THREAD_FUNNELED`: for multi-threaded MPI processes, when MPI calls are done by the main thread only.
- `MPI_THREAD_SERIALIZED`: for multi-threaded MPI processes, when only one thread at a time can call MPI.
- `MPI_THREAD_MULTIPLE`: for multi-threaded MPI processes, when multiple threads can call MPI concurrently.

X10 places are multi-threaded, and all the threads can invoke communication actions. Thus only `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE` are feasible for X10. X10RT-MPI uses `MPI_THREAD_MULTIPLE` by default, unless the MPI implementation does not support it or the user explicitly chooses the serialized mode. Due to lack of stable support for `MPI_THREAD_MULTIPLE` in current MPI-ULFM implementations, we changed the default behaviour of X10RT-MPI to use the serialized threading mode for any MPI-ULFM implementation. After a successful initialization, the default communicator `MPI_COMM_WORLD` is used throughout the execution of the X10 program.

3.2.2 Active Messages

An active message encapsulates a function to be executed at a destination process. The **`at`** construct is used to send active messages in X10. For example, **`at`** `(q)` **`async`** `S;` sends an active message to execute statement `S` at place `q` asynchronously. X10 relies on X10RT to perform the required communication for exchanging active messages.

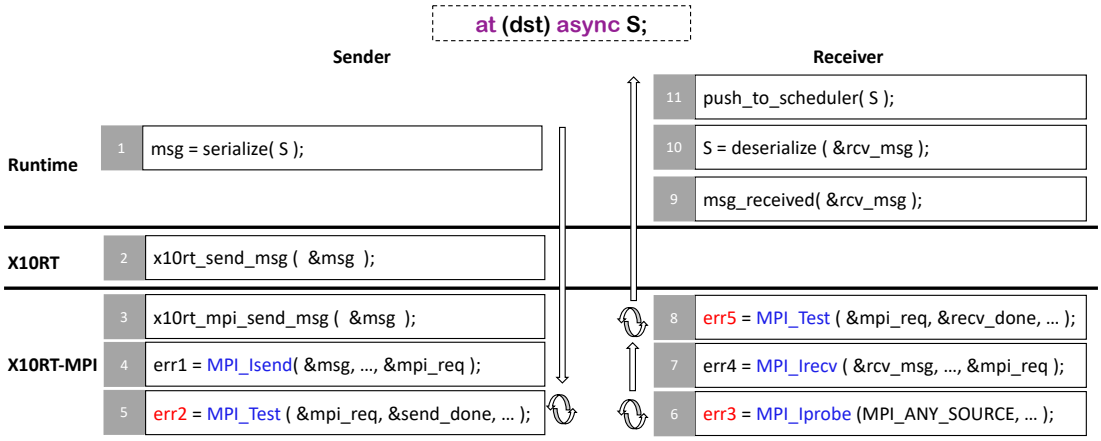

Figure 3.2: X10 active messages implementation over MPI. Some of the method names shown in the figure are slightly different from the actual implementation for more clarity. The error variables colored in red, at steps 5, 6, and 8, mark the points at which MPI-ULFM may notify process failures, as will be discussed in Section 3.3.2.

Figure 3.2 shows the flow of actions by the X10 runtime and its MPI transport layer for sending an active message. On invoking **`at`** `(q)` **`async`** `S;`, the active message `S` is serialised into a send buffer named `msg` that is then passed to the abstract method `x10rt_send_msg(&msg)` in X10RT. This method invokes the corresponding concrete implementation `x10rt_mpi_send_msg(&msg)`, which uses MPI to send the buffer `msg`.

X10 uses non-blocking two-sided communication operations for transferring data and active messages between places. The sender place invokes `MPI_Isend(..)` and passes, among other parameters, the send buffer `msg` and a handle named `mpi_req` that can be used for checking the status of this send request (see Figure 3.2-step 4). MPI applications can use `MPI_Wait(..)` to block a process that issued a non-blocking

request until the request completes. However, as mentioned in Section 3.1, blocking an X10 place may lead to deadlocks. Therefore, X10 uses the alternative non-blocking method `MPI_Test(..)` to check the completion status of its send and receive requests given their handles. Each place maintains a queue of pending requests and periodically checks their completion status. In Figure 3.2, the sender repeatedly calls `MPI_Test(..)` in step 5 to check the completion status of the send request issued in step 4. When the send request completes, X10RT-MPI removes its handle from the pending requests queue and expects the message to eventually reach its destination.

Each X10 place actively probes for incoming messages from other places by repeatedly calling the non-blocking function `MPI_Iprobe(..)` and passing the wildcard value `MPI_ANY_SOURCE` as the source parameter (step 6). When `MPI_Iprobe` detects the arrival of a message, X10RT-MPI extracts the message's metadata including its source and number of bytes and then invokes the non-blocking request `MPI_Irecv(..)` to receive the message (step 7). A handle to this receive request is stored in the pending requests queue and is periodically checked for completion using `MPI_Test(..)` (step 8). When the message is received, `X10RT-MPI` passes it to the X10 runtime by calling `msg_received(&msg)` (step 9). The message is then deserialised to create the activity `S` (step 10), which is finally pushed to the work-stealing scheduler of X10 to eventually execute it (step 11).

It is worth noting that MPI has its own definition of what makes a request complete. A receive request is considered complete after the message is fully received. However, a send request is considered complete when the application can reuse the send buffer. If the send buffer is relatively small, MPI may copy it to an internal buffer and declare the request complete before it starts sending the message. If the send buffer is beyond MPI's capacity, MPI will not be able to copy it to an internal buffer; it will not declare the request complete until after the message is actually sent to prevent the user from altering the send buffer. In the case of buffering, the source place may fail after declaring a send request complete but before sending the message. This possibility is taken into consideration in RX10. The protocols used for tracking the active messages assume that a message may be lost due to the failure of its source or its destination. These protocols will be described in details in Chapter 4.

To summarise, X10 implements active messages using non-blocking MPI functions (`MPI_Isend`, `MPI_Irecv`, and `MPI_Test`). Each place is actively scanning its inbox for incoming messages from other places using `MPI_Iprobe(MPI_ANY_SOURCE,..)`.

3.2.3 Team Collectives

The APGAS model of X10 is well-suited for bulk-synchronous SPMD programs that rely on collective operations for synchronization or data sharing. The following text quoted from [Hamouda et al., 2016] briefly describes how the X10 team implemented collective operations in X10. It refers to two types of MPI collective functions: blocking and non-blocking. A blocking collective function blocks the calling thread until the collective operation completes. A non-blocking collective function returns after initiating the collective operation without waiting for the operation to complete.

> X10 contains a collective API similar to MPI, located in `x10.util.Team`, offering collective operations such as barrier, broadcast, all-to-all, etc. `Team` attempts to make use of any collective capabilities available in the underlying transport. For transports that provide native collectives, `Team` maps its operations to the transport collective implementations. For transports that do not provide collectives, such as TCP/IP sockets, `Team` provides *emulated collective implementations*. An interesting combination arises when the underlying transport supports some, but not all, of the functionality needed by X10. The X10 thread model requires non-blocking operations from the network transport, because there may be runnable tasks in the thread's work queue, and a blocking network call will prevent that runnable work from completing, leading to possible deadlock. MPI-3 offers non-blocking collectives, but other than barrier these are optional, and MPI-2 only supports blocking collectives. For best performance we still want to make use of these, so our implementation calls an emulated barrier immediately before issuing the blocking collective. This allows us to line up all places so that when they reach the blocking operation, they are all in a position to pass through the collective immediately. [Hamouda et al., 2016]

Figure 3.3 shows an example program using `Team` for a reduction operation and the three different `Team` implementations: emulated, native blocking and native non-blocking.

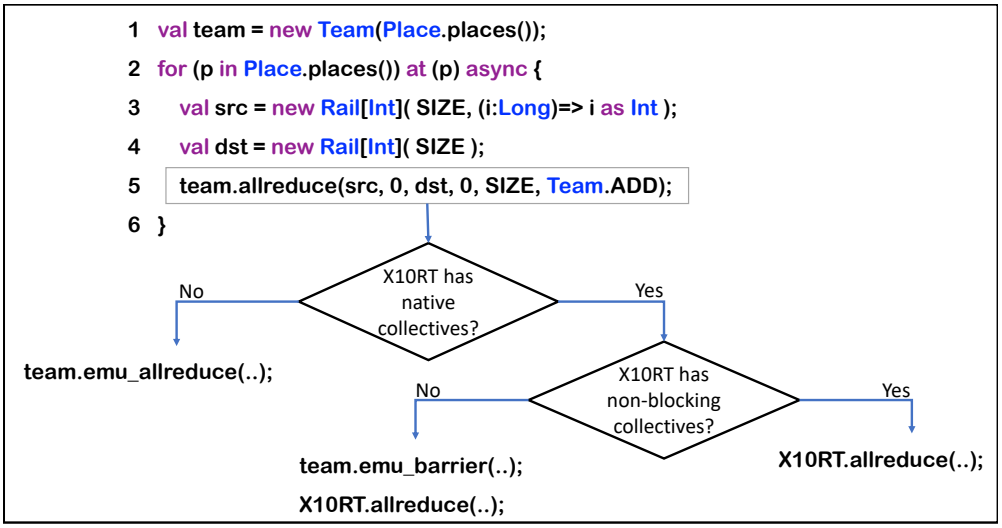

Figure 3.3: Team API implementation options.

3.2.3.1 The Emulated Implementation

The emulated `Team` implementation organizes the places in a binary tree that facilitates recursive communication between the nodes in a unidirectional way (e.g. for broadcast and gather) or a bidirectional way (e.g. for barrier and allreduce). The collective starts by blocking the parent places in a busy waiting loop until they receive upward notifications from their children, as shown in Line 2 of the pseudocode below. When a parent receives notifications from its children, it switches to the child role and passes the notification to its own parent. A child place notifies its parent by creating a **`finish`** that governs a remote asynchronous task at the parent place (Line 4), which serves two purposes. First, it notifies the parent that the child has entered the collective (Line 5), which implies that the places in the child's sub-tree have also entered the collective. Second, it blocks the child until the upper part of the tree enters the collectives (Line 6). The completion of the **`finish`** returns control back to the child indicating the completion of the upward notification phase. At this point, the child switches to the parent role and releases its own children who are blocked *locally* in Line 6 waiting for the downward notification. If the collective requires data to be transferred, these may be piggybacked on the notification messages.

```
1 /*as a parent:*/
2 wait_for_upward_notification();
3 /*as a child:*/
4 finish at (parent) async  {
5     deliver_upward_notification();
6     wait_for_downward_notification();
7 }
8 /*as a parent:*/
9 deliver_downward_notification();
```

3.2.3.2 The Native Implementation

When `Team` is used in native mode, it implements the collective interfaces by delegating to the underlying MPI implementation. `Team` surrounds each call to a native MPI collective with a local **`finish`** that serves one purpose — ensuring the completion of the native collective that may be non-blocking. It emulates the execution of the MPI collective as if it is done by an **`async`** statement. It does so by first calling a runtime function that increments the number of **`finish`** tasks, then calling the X10RT-MPI function that invokes the MPI collective. X10RT-MPI receives a reference to the surrounding **`finish`** and uses it to notify the termination of the emulated **`async`** statement after the MPI collective completes. A notification of a blocking MPI collective is sent immediately after the blocking call returns. On the other hand, a non-blocking collective is notified only after `MPI_Test` reports its completion, in the same way active messages are checked for completion (see Section 3.2.2). The same mechanism is applied with the PAMI transport.

3.3 MPI-ULFM Overview

As high performance computing applications run longer on larger numbers of cores, they are more likely to experience process failures. If HPC programmers do not prepare their applications to deal with process failures, much time and energy will be wasted on supercomputers due to restarting failed computations. MPI-ULFM addresses this problem by extending MPI with fault tolerance features that enable programmers to design a variety of failure recovery mechanisms. It focuses on *fail-stop* failures in which a process impacted by a hardware or a software fault crashes or stops communicating.

The following paragraph quoted from the the ULFM specification document [MPI-ULFM, 2018] describes its main design principles at a high level:

> The expected behaviour of MPI in the case of an MPI process failure is defined by the following statements: any MPI operation that involves a failed process must not block indefinitely but either succeed or raise an MPI error ...; an MPI operation that does not involve a failed process will complete normally, unless interrupted by the user through provided functionality. Errors indicate only the local impact of the failure on an operation and make no guarantee that other processes have also been notified of the same failure. Asynchronous failure propagation is not guaranteed or required, and users must exercise caution when determining the set of processes where a failure has been detected and raised an error. If an application needs global knowledge of failures, it can use the interfaces defined in Section ... to explicitly propagate the notification of locally detected failures. [MPI-ULFM, 2018]

In this section we describe ULFM's features in more detail, focusing on the aspects most relevant to a task-based runtime system like X10. Because X10 does not use MPI's RDMA operations, we do not cover those operations in this thesis.

3.3.1 Fault-Tolerant Communicators

A communicator in MPI represents a group of processes that are allowed to communicate with each other. The main communicator `MPI_COMM_WORLD` includes all the created processes at application start-up. By default, MPI communicators are not fault-tolerant. When the first error arises, whether it is a memory error, a communication error or otherwise, the runtime system immediately halts. That is because all MPI communicators use `MPI_ERRORS_ARE_FATAL` as a default error handler. MPI provides a second error handler called `MPI_ERRORS_RETURN` which, instead of halting the runtime system, reports the error to the calling process. However, for this error handler to be useful for recovering from process failures, the behaviour of the runtime system after a process failure must be specified.

ULFM closes the failure specification gap in MPI so that communicator error handlers start to have a practical value for fault tolerance. By assigning an error

handler to the communicator, ULFM can reliably report process failure errors to active processes and permit them to continue communicating according to precisely defined rules. The following three error classes are added in ULFM for reporting process failures:

- `MPI_ERR_PROC_FAILED_PENDING`: for non-blocking receive operations where the source process is `MPI_ANY_SOURCE`, this error indicates that no matching send is detected yet, but one potential sender has failed.
- `MPI_ERR_PROC_FAILED`: for all communication operations, this error indicates that a process involved in the communication request has failed.
- `MPI_ERR_REVOKED`: for all communication operations, this error indicates that the communicator has been invalidated by the application.

3.3.2 Failure Notification

ULFM specifies the conditions that lead to notifying a process of the failure of another process. Processes that may be notified of a failure are those communicating with a dead process directly (through point-to-point operations) or indirectly (through collectives). When a failure is detected by one process, it is not propagated by default to other processes. Unless propagating the notification is requested explicitly by one process, failure knowledge remains local and may differ between processes. The following are the main rules governing failure notification for non-blocking operations, collectives, and the use of `MPI_ANY_SOURCE`.

When non-blocking communication is used, failure notification is deferred from the initiation functions, such as `MPI_Isend` or `MPI_Iallreduce`, to the completion functions `MPI_Test` and `MPI_Wait`. For example, in Figure 3.2, we marked the return values that can notify failures in red color. If the source place is dead, ULFM will not notify X10 with that failure in step 7 where `MPI_Irecv` is called. However, it will notify it in step 8 when `MPI_Test` is called.

ULFM provides non-uniform failure reporting for the collective operations [Bland et al., 2013]. Depending on the collective implementation, when a process dies, some processes may report the collective as successful, while others may report it as failed.

The wildcard value `MPI_ANY_SOURCE` is useful for detecting failures of any process in a communicator. When invoking a receive or a probe operation from `MPI_ANY_SOURCE`, an error is raised when MPI detects the failure of any potential sender.

3.3.3 Failure Mitigation

The user-level resilience model of MPI-ULFM provides great flexibility for application recovery. Rather than transparently recovering applications by a single fault tolerance mechanism, ULFM provides a few failure mitigation functions that can be integrated in different ways to implement different recovery mechanisms. We show these functions in Table 3.1.

Table 3.1: MPI-ULFM failure mitigation functions.

	Operation	Type	Description
1	MPI_COMM_FAILURE_ACK(comm)	local	acknowledges the notified failures
2	MPI_COMM_FAILURE_GET_ACKED(comm, grp)	local	returns the group of acknowledged failed processes
3	MPI_COMM_REVOKE(comm)	remote	invalidates the communicator
4	MPI_COMM_SHRINK(comm,newcomm)	collective	creates a new communicator by excluding dead processes from the original communicator
5	MPI_COMM_AGREE(comm,flag)	collective	provides a fault-tolerant agreement algorithm

When a process is notified of a failure, ULFM expects that some applications might need to silence repeated notifications of the same failure. The first function `MPI_COMM_FAILURE_ACK` is added for that purpose. It enables a user to acknowledge the failures notified by a communicator so far. After acknowledging a failure, receive operations that use `MPI_ANY_SOURCE` and the new collective operations `MPI_COMM_SHRINK` and `MPI_COMM_AGREE` will proceed successfully even if they involve the failed process. The second function `MPI_COMM_FAILURE_GET_ACKED` is used for identifying the failed processes that were acknowledged. Figure 3.4 shows a graphical example on these two functions. In this example, two ranks have failed, rank 1 followed by rank 3. The failure of rank 1 was acknowledged by calling `MPI_COMM_FAILURE_ACK`, but the failure of rank 3 was not acknowledged. As a result, a call to `MPI_COMM_FAILURE_GET_ACKED` following the failure of rank 3 returns only rank 1.

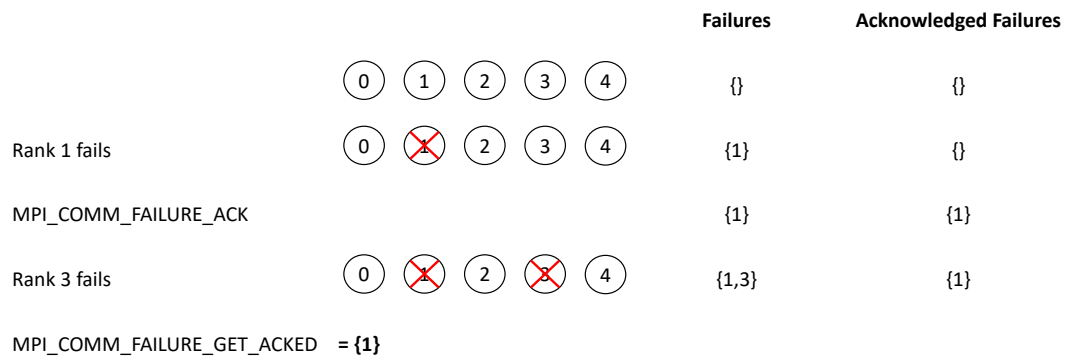

Figure 3.4: Use of MPI_COMM_FAILURE_ACK and MPI_COMM_FAILURE_GET_ACKED for failure detection and acknowledgement.

The third function `MPI_COMM_REVOKE` is added for explicit global failure propagation. ULFM does not implicitly propagate failure notifications to all processes in a communicator. Thus, processes that do not communicate with the failed process will not detect its failure as long as the communicator is valid. If failure propagation is needed, an application may call `MPI_COMM_REVOKE` to invalidate the communicator. After one process revokes a communicator, all processes sharing the same communicator will receive `MPI_ERR_REVOKED` when they call any non-local MPI operation, except `MPI_COMM_SHRINK` and `MPI_COMM_AGREE`. These processes must recover the communicator to resume execution.

Communicator recovery strategies are generally classified as shrinking or non-shrinking (see Section 2.2.3.1 for the description of these classes). In shrinking recovery, a new communicator is created by excluding the dead processes in a given communicator. The fourth function `MPI_COMM_SHRINK` is added for that purpose. Non-shrinking recovery requires replacing dead processes with new ones, so that the application can restore its state using the same number of processes. The MPI-3 standard, on which MPI-ULFM is based, supports dynamic process creation using `MPI_COMM_SPAWN`. Thus by combining `MPI_COMM_SHRINK` and `MPI_COMM_SPAWN`, MPI applications can implement non-shrinking recovery methods [Ali et al., 2014].

Figure 3.5 demonstrates the use of `MPI_COMM_REVOKE` and `MPI_COMM_SHRINK` for performing shrinking recovery. This example presents a 6-rank communicator, where each rank is communicating with only one other rank. When rank 1 failed, only rank 0 can detect the failure. However, rank 0 cannot shrink the communicator alone; the collective function `MPI_COMM_SHRINK` requires the participation of every non-failed rank in the communicator. By calling `MPI_COMM_REVOKE`, rank 0 invalidates the communicator, not only for itself but for all the other ranks. In this example, each rank handles an invalid communicator by calling `MPI_COMM_SHRINK`. This results in creating a 5-rank communicator for resuming the application.

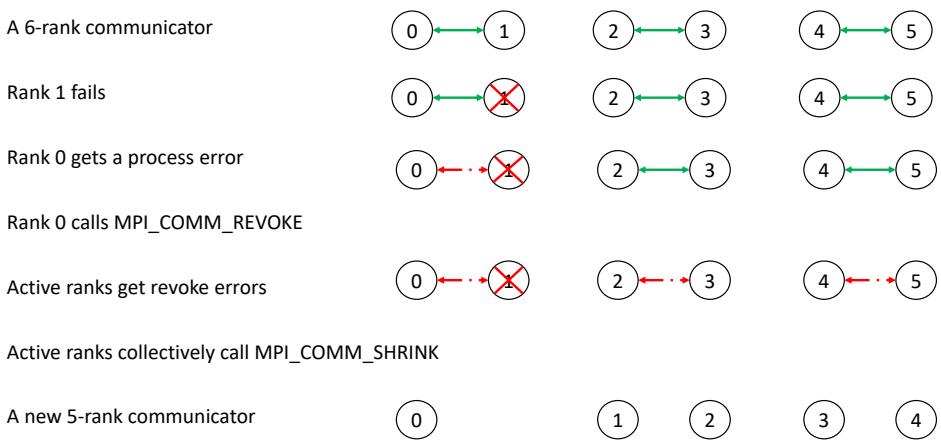

Figure 3.5: Use of MPI_COMM_REVOKE and MPI_COMM_SHRINK to perform shrinking recovery. Here we assume that communication is performed between pairs of processes.

The fifth and last function `MPI_COMM_AGREE(comm,flag)` provides a fault-tolerant agreement algorithm. It is a collective operation that results in having all the active processes agreeing on the value of the integer in/out parameter flag and also implicitly agreeing on the group of failed processes. The agreed value is the bitwise-and of the values provided by all active processes. While normal collectives may succeed at some processes and fail at others, ULFM guarantees that `MPI_COMM_AGREE` will either complete successfully at all the processes or fail at all the processes.

This concludes our overview of the MPI-ULFM specification. We find MPI-ULFM promising for bridging the gap between performance and resilience. By providing user-controlled resilience in a performance-oriented programming model, many optimizations can be leveraged for handling failures more efficiently at scale.

3.4 Resilient X10 over MPI-ULFM

MPI-ULFM provides a flexible framework for HPC users to learn and apply different fault tolerance concepts in their applications. However, the low-level programming model of MPI increases the programming cost for achieving that goal. Arguably, MPI-ULFM can play a more effective role in making fault tolerant programming more accessible to users by being a compilation target for resilient high-level languages. In this section, we demonstrate this approach in the context of the X10 language.

3.4.1 Resilient X10 Transport Requirements

The minimalist approach to resilience adopted by X10 reduces the resilience requirements needed from the underlying transport runtime. In fact, RX10 requires only two mandatory features from X10RT to enable applications to recover from failures correctly:

1. allowing non-failed processes to communicate.
2. allowing a place to eventually detect the failure of any other place even without direct communication with the failed place. In other words, RX10 requires *global failure detection*. This requirement is primarily needed to enable the **`finish`** construct to detect the failure of directly and indirectly generated remote tasks within its scope.

Other transport features that are desired but not mandatory are:

3. providing fault-tolerant collectives to speed up `Team` operations.
4. supporting dynamic process creation for non-shrinking recovery.

From the previous section, we conclude that MPI-ULFM provides all the mandatory and desired features needed by RX10. It also provides an additional feature, fault-tolerant agreement, that resilient applications can leverage for failure recovery.

3.4.2 Global Failure Detection

One way to provide global failure detection with ULFM is by revoking the communicator. However, this method is expensive since it requires, in addition to the cost of propagating the failure to other places, the cost of collectively reconstructing a new communicator with every failure.

The other option that is directly applicable to X10 is using `MPI_ANY_SOURCE`. According to ULFM's semantics, an operation that uses `MPI_ANY_SOURCE` will raise an error when MPI detects the failure of any potential sender (unless the failure has been acknowledged). Each X10 place is actively probing the network for incoming messages from any other place by calling `MPI_Iprobe (MPI_ANY_SOURCE, ..)`, as shown in Figure 3.2. As a result, each X10 place will eventually be notified of the failure of any other place. This feature enabled us to avoid repairing the communicator with every failure, while meeting the requirement of providing global failure detection to RX10.

3.4.3 Identifying Dead Places

In resilient mode, X10RT is expected to detect communication errors and to use them to identify dead places. The design of RX10 requires each resilient implementation of X10RT to implement the following two interfaces that the runtime uses to discover the detected failures:

- `int x10rt_ndead ()`: returns the number of dead places.
- `bool x10rt_is_place_dead (int p)`: checks whether a place is dead.

Listing 3.1: MPI error handler for Resilient X10.

```
1 //global state
2 int ndead = 0;    //number of dead places
3 int* dead_places; //array of dead places ranks
4
5 void mpiCustomErrorHandler(MPI_Comm* comm, int *errorCode, ...) {
6     MPI_Group f_group; int f_size;
7
8     //Acknowledge & query failed processes
9     MPI_Comm_failure_ack(*comm);
10    MPI_Comm_failure_get_acked(*comm, &f_group);
11    MPI_Group_size(f_group, &f_size);
12    int* f_ranks = malloc(...);
13    MPI_Group_translate_ranks(f_group,... ,f_ranks);
14
15    //Update global state
16    dead_places = f_ranks;
17    ndead = f_size;
18 }
```

To detect dead places, we registered the error handler in Listing 3.1 to the default communicator `MPI_COMM_WORLD`. The error handler uses local ULFM operations to acknowledge and query the list of dead places (Lines 9–13). This list is stored in the transport's global state (Lines 16–17) which is used for implementing the two query functions listed above. Different places can have different knowledge about the status of other places which is permitted in RX10. X10RT-MPI serializes calls to MPI via a lock which results in executing the error handler atomically (see Section 3.2.1).

3.4.4 Native Team Collectives

When using MPI for communication, each `Team` is mapped to a sub-communicator of `MPI_COMM_WORLD` that is used for calling MPI collective operations only. Providing fault-tolerant `Team` operations on top of MPI-ULFM mainly impacted the process of `Team` creation and failure notification.

3.4.4.1 Team Construction

The interfaces for creating an X10 `Team` and creating an MPI communicator have a major difference. `Team` creation is exposed as a centralized single-place function, whereas communicator creation in MPI is a collective function. When X10 uses MPI, the constructor of the `Team` class implicitly sends MPI messages to all the places to trigger the collective invocation of `MPI_COMM_CREATE`. This function is used to create a sub-communicator for the new `Team` from the parent communicator `MPI_COMM_WORLD`. The root place constructing the `Team` waits for an acknowledgement message from each place to detect the success or failure of communicator construction. A failure at any place results in throwing an exception to the application.

According to the ULFM specification, if some of the communicator processes failed, collectives over that communicator may fail at some of the participating processes and succeed at others. As a result, if one X10 place failed, `MPI_COMM_WORLD` would not be useful for constructing any new `Team` because `MPI_COMM_CREATE` would never fully succeed over `MPI_COMM_WORLD`. Repairing `MPI_COMM_WORLD` by revoking and shrinking it after every failure would solve this problem, but at unacceptable performance cost for applications that do not use `Team`.

We solve this problem by performing a communicator repair step every time a new `Team` is constructed. While this increases the cost of creating a `Team`, this cost is negligible as X10 programs often create one `Team` object throughout their execution. The solution is as follows: we use `MPI_COMM_SHRINK` to create a new temporary communicator that excludes any dead process in `MPI_COMM_WORLD`. The temporary communicator is then used to create the `Team`'s communicator, as shown in the following code:

```
1 MPI_Comm team_comm, shrunken;
2 MPI_COMM_SHRINK(MPI_COMM_WORLD, &shrunken);
3 MPI_COMM_CREATE(shrunken, group, &team_comm);
```

One vulnerability that the code above does not handle is the failure of a process in the `shrunken` communicator between executing Line 2 and Line 3. This failure may still cause `MPI_COMM_CREATE` to succeed at some processes, but fail at others. Calling ULFM's `MPI_COMM_AGREE` function can be useful to decide whether or not all processes succeeded in creating the communicator. However, we did not need this agreement call thanks to the centralized nature of `Team` creation in X10. The root place will notify the failure of `Team` creation if any of the places failed in completing the above code snippet.

Note that the most recent release of X10 at the time of writing, X10 2.6.1, follows the construction of the communicator with an internal fan-out **`finish`** that notifies each place with its order among the `Team` places. This step results in additional cost in resilient mode for tracking the termination of this fan-out finish resiliently. Sharing the place's order in the `Team` can be done more efficiently by piggybacking this information in the same X10RT-MPI messages that orchestrate `Team` creation or by adding the list of `Team` members as one of the object identifiers. For simplicity, we implemented the latter method in our X10 repository. In X10 2.6.1, passing a `Team` object to another place results in passing a small message containing the `Team` ID only, whereas in our implementation passing a `Team` object results in passing a bigger message containing the `Team` ID and an array of N values of type `Long` representing the list of places. By knowing the list of places, a place can calculate its order in the `Team` without additional communication. By avoiding the broadcast messages at the time of `Team` creation, we also avoid the overhead of the resilient **`finish`** protocol and make the performance of `Team` creation virtually equal in both X10 and RX10.

3.4.4.2 Team Failure Notification

As described in Section 3.2.3.2, `Team` surrounds each call to a native MPI or PAMI collective with a local **`finish`** that waits for the completion of the underlying native collective. We leveraged the finish construct's failure notification support to allow `Team` operations to notify process failure errors. When the native MPI collective failure raises a process failure error, we add a `DeadPlaceException` to the finish block before notifying the completion of the collective. Finish at the places experiencing the failure will throw this exception to the calling task. Other places may complete successfully. If successful places call another collective operation, this collective call will hang because the places that detected the failure may not participate in it. To avoid this problem, we revoke the `Team`'s communicator upon detecting a collective failure at any place so that future `Team` operations fail at all participating places.

Emulated `Team` collectives employ a simple error propagation mechanism to avoid deadlocks due to non-uniform error reporting. A place that detects a failure of its parent invalidates itself and sends asynchronous messages to invalidate its grandparent, its children, and grandchildren. The invalidation messages propagate to the top and bottom of the collective tree until all members are notified that the `Team` object is no longer valid.

3.4.4.3 Team Agreement

Reaching agreement between processes as they switch between phases is often required in SPMD applications. Agreement algorithms are also useful for failure recovery, as they enable surviving distributed processes to agree on the state of the system after unexpected failures [Herault et al., 2015]. X10 and ULFM provide different methods for facilitating agreement.

The **finish** construct in X10 can be used as a centralized agreement coordinator as shown in Listing 3.2. Running multiple phases requires creating a fan-out **finish**, i.e. a finish that forks a remote task at each place, for each phase. Finish supervises the execution of one phase and reports any detected place failures. All places pass their results to the **finish** home place in which agreement can be decided by comparing the results. Unfortunately, performing phase agreement using a fan-out finish causes high performance overhead for resilient applications because tracking the creation and termination of remote tasks comes with additional communication cost in resilient mode. We explain and quantify the resilience overhead of a fan-out finish in Section 4.11 in the next chapter.

Listing 3.2: Centralized agreement between three phases using finish (failure handling is not shown for simplicity). We assume having a `PhaseResult` class with two methods: `addResult` which reports the phase result of one place and `agree` which decides if the results of all places are the same then deletes the results.

```
1 //gr used for collecting the results of all places at each phase
2 val gr = GlobalRef[PhaseResult](new PhaseResult());
3 finish for (p in places) at (p) async {
4     val a = phase1();
5     //send the result to the home place
6     at (gr) async gr().addResult(a);
7 }
8 if (gr().agree()) {
9     print("phase 1 agreement reached");
10    finish for (p in places) at (p) async {
11        val b = phase2();
12        at (gr) async gr().addResult(b);
13    }
14    if (gr().agree()) {
15        print("phase 2 agreement reached");
16        finish for (p in places) at (p) async {
17            val c = phase3();
18            at (gr) async gr().addResult(c);
19        }
20        if (gr().agree()) {
21            print("phase 3 agreement reached");
22        }
23    }
24 }
```

On the other hand, ULFM supports distributed agreement using the collective function `MPI_COMM_AGREE`. Coordinating three phases of execution, such that agreeing on the state of the previous phase is established before starting the next phase, can be done using one fan-out finish with longer-lived tasks as shown in Listing 3.3. By avoiding the unnecessary creation of new remote tasks at each phase, resilient applications can execute more efficiently. To enable X10 programmers to exploit distributed agreement in their applications, we added the function `Team.agree(int)` which directly maps to `MPI_COMM_AGREE`. The integer parameter can represent a phase result, the number of dead places known to a place, or whatever value all places need to agree on. If one place failed, `Team.agree(int)` would fail at all `Team` members — a feature that is not guaranteed in any other collective function. In Section 5.3.2.2, we show how we used this function to reduce the checkpointing overhead of the iterative application framework.

Listing 3.3: Distributed agreement between three phases (failure handling is not shown for simplicity).

```
1 val team = new Team(places);
2 finish for (p in places) at (p) async {
3     val a = phase1();
4     if (a == team.agree(a)) {
5         print("phase 1 agreement reached");
6         val b = phase2();
7         if (b == team.agree(b)) {
8             print("phase 2 agreement reached");
9             val c = phase3();
10            if (c == team.agree(c)) {
11                print("phase 3 agreement reached");
12            }
13        }
14    }
15 }
```

3.4.5 Non-Shrinking Recovery

Our implementation did not use the feature of dynamically creating MPI processes to replace dead places with new ones. However, resilient X10 applications can still perform non-shrinking recovery by pre-allocating spare processes at application start time. Bungart and Fohry [2017a] extended our implementation by supporting dynamic process creation over MPI-ULFM. However, their implementation is not publicly available yet.

3.5 Performance Evaluation

The objective of this section is to demonstrate the efficiency and scalability advantages that resilient X10 applications can achieve by switching from the socket transport to the MPI-ULFM transport. We give more focus on the performance of the `Team` APIs because collective communication is a key mechanism in the widely used bulk-synchronous model. `Team` also gives us the flexibility to exercise both MPI's point-to-point interfaces as well as its collective interfaces through the emulated and native `Team` implementations, respectively.

The following experiments aim to answer the following questions:

1. What is the failure-free resilience overhead of MPI-ULFM?
2. What is the performance advantage of using MPI-ULFM as a transport layer for RX10 applications as opposed to using the sockets transport layer?
3. What is the performance advantage of replacing the emulated collectives with native MPI collectives for RX10 applications?

3.5.1 Experimental Setup

We conducted the following experiments on the *Raijin* supercomputer at NCI, the Australian National Computing Infrastructure, and on on a cluster of 30 virtual machines from the Australian *NECTAR* research cloud. Each of the virtual machines has 12 virtual cores, providing a total of 360 virtual cores for our experiments.

Each compute node in *Raijin* has a dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors and connects to the other nodes using an Infiniband FDR network. We configured our jobs to allocate 10 GiB of memory per node. Our *NECTAR* virtual machines are hosted at the NCI zone and were installed using the 'm2.xlarge' template, which allocates 45 GiB of memory and 12 virtual cores for each virtual machine.

Our MPI jobs on both platforms statically bound each process to a separate core. Our X10 jobs allocate one place per core and oversubscribe each core with two internal place threads: a worker thread[1] for handling user tasks and an immediate thread[2] for handling low-level runtime communication. Although an X10 place can be mapped to an entire node, the microbenchmarks used in this chapter create one task per place. Therefore, using one core per place is sufficient.

The reference implementation of MPI-ULFM used across the whole thesis is ULFM-2. When we mention ULFM without any version number, we refer to the ULFM-2 implementation. ULFM was built from revision `e87f595` of the `master` branch of the repository at `https://bitbucket.org/icldistcomp/ulfm2.git`. 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.

[1] One worker thread is configured by setting the environment variable X10_NTHREADS=1

[2] One immediate thread is configured by setting the environment variable X10_NUM_IMMEDIATE_THREADS=1

Each of the experiments below was conducted 30 times. The tables report the median value in black color, and the 25th and 75th percentiles within brackets in gray color.

3.5.2 Performance Factors

In order to better understand the performance results in this section, we explain two important factors that impact the performance of our experiments.

3.5.2.1 The Immediate Thread

The X10 language provides the `@Immediate` annotation to give higher priority to certain **`async`** statements. An immediate **`async`** is expected to execute small non-blocking tasks, mostly used for low-level runtime communications. A special thread, called the immediate thread, is created by the scheduler to continuously probe for incoming immediate tasks and to execute them in order. While an immediate thread is limited to executing immediate tasks only, a worker thread can execute normal and immediate tasks.

X10RT-Sockets implements the immediate thread more efficiently than X10RT-MPI. X10RT-Sockets *suspends the immediate thread* until an immediate task is received using the blocking `poll()` interface from the `pthreads` library. On the other hand, X10RT-MPI uses *a busy waiting loop for the immediate thread* to continuously scan the network for incoming immediate tasks. Although MPI provides a blocking `MPI_PROBE` interface, this interface is not safe for X10 to use with the `MPI_THREAD_SERIALIZED` mode. This mode serializes the communication calls from the different threads, thus blocking one thread will prevent others from progressing. Unfortunately, this busy waiting behaviour may slow down the progress of the worker threads, especially when the immediate thread competes with the worker threads over the same core. Starting the X10 places without any immediate thread over MPI is mostly safe for non-resilient executions. However, the absence of the immediate thread can cause the resilient executions to deadlock because RX10 makes extensive use of immediate tasks for the resilient **`finish`** protocol. For fair comparison between X10 and RX10, we start all X10 places with one worker thread and one immediate thread.

3.5.2.2 Emulated Team versus Native Team

As described in Section 3.2.3 and Section 3.4.4, the `Team` class is designed to use the collective interfaces provided by the underlying transport layer. For transport layers that don't provide collective interfaces, `Team` provides emulated collectives. The emulated collectives exchange active messages between the X10 places — using **`finish`**, **`at`**, and **`async`** — to achieve their intended goal. On the other hand, the native collectives delegate their functions to the MPI layer, thereby avoiding the need for sending or tracking active messages by the X10 runtime. We expect this difference to be influential in RX10, because tracking the remote active messages with a resilient **`finish`** requires additional control messages compared to a non-resilient **`finish`**,

which is the main cause for the resilience overhead in X10. Therefore, it is expected that the emulated collectives will be sensitive to the resilience overhead of **`finish`**, while the native collectives won't. The native collectives are, however, expected to be sensitive to the resilience overhead of ULFM.

3.5.3 MPI-ULFM Failure-Free Resilience Overhead

Fault tolerance may be expected to come at a performance cost. When we use ULFM as a transport layer for X10, any resilience overhead imposed by ULFM will be inherited by all X10 applications. To study the resilience overhead of ULFM, we compared its performance to 1) ULFM without fault tolerance and to 2) Open MPI v3.0.0. ULFM without fault tolerance is built from the ULFM source code using the configuration option `--without-ft`. This option disables the code paths related to fault tolerance in ULFM and therefore compiles an almost standard Open MPI implementation. Open MPI v3.0.0 (OMPI for short) is a standard release of Open MPI that does not provide any fault tolerance support. We choose this version because its release date is the closest to the ULFM revision that we used. Table 3.2 shows the performance of three `Team` operations, barrier, broadcast, and allreduce, over 360 places running on *Raijin*.

`Team` collectives are used as microbenchmarks since they can be configured to activate either MPI collectives or MPI point-to-point communications. MPI point-to-point communications can be activated by forcing `Team` to use the emulated collectives despite the availability of native collectives, by setting the environment variable `X10RT_X10_FORCE_COLLECTIVES`=1.

Since we are concerned with the communication overhead rather than the memory management overhead, we use a small array of 2048 doubles in the broadcast and allreduce collectives. The array size (16384 bytes) exceeds MPI's eager limit (12288 bytes) on both *Raijin* and *NECTAR* platforms, therefore sending messages is performed using the rendezvous protocol rather than the eager protocol on both systems [Squyres, 2011].

The first three rows in Table 3.2 show the performance of the three collectives in pure MPI benchmarking programs. The programs invoke the non-blocking collectives: `MPI_Ibarrier`, `MPI_Ibcast`, and `MPI_Iallreduce` followed by a busy waiting loop that checks the completion of the collective call using `MPI_Test`. That is, in principle, the same mechanism applied by X10RT-MPI for executing `Team` collectives natively.

The bare MPI performance of each of the collectives is indifferentiable with ULFM and ULFM without fault tolerance, indicating the absence of a measurable resilience overhead by ULFM in failure-free collective executions. OMPI's barrier performance is comparable to the ULFM implementations; however, its bcast and allreduce performance is slower. This could result from unavoidable implementation differences between the source code of OMPI and ULFM due to their independent parallel development. The variability of the results, as shown in the side brackets, is reasonable using the three MPI implementations directly.

Table 3.2: Performance of Team collectives with different MPI implementations on Raijin with 360 places.

Runtime	Transport	Team Impl.	barrier	bcast	allreduce
Pure MPI	OMPI	-	0.1 ms $[^{00.1}_{00.1}]$	0.6 ms $[^{00.8}_{00.5}]$	2.8 ms $[^{02.9}_{02.8}]$
	ULFM without FT	-	0.1 ms $[^{00.2}_{00.1}]$	0.1 ms $[^{00.1}_{00.1}]$	0.4 ms $[^{00.4}_{00.4}]$
	ULFM	-	0.1 ms $[^{00.2}_{00.1}]$	0.1 ms $[^{00.2}_{00.1}]$	0.4 ms $[^{00.4}_{00.4}]$
X10	OMPI	Native	5.6 ms $[^{15.2}_{01.6}]$	0.6 ms $[^{00.7}_{00.6}]$	3.0 ms $[^{03.0}_{02.9}]$
	ULFM without FT	Native	8.2 ms $[^{16.3}_{00.7}]$	0.1 ms $[^{00.1}_{00.1}]$	6.7 ms $[^{16.0}_{01.2}]$
	ULFM	Native	7.6 ms $[^{14.8}_{02.4}]$	9.6 ms $[^{12.4}_{05.2}]$	9.1 ms $[^{14.7}_{03.4}]$
	OMPI	Emulated	81.5 ms $[^{87.5}_{77.3}]$	89.6 ms $[^{99.5}_{81.5}]$	80.5 ms $[^{89.1}_{78.7}]$
	ULFM without FT	Emulated	76.0 ms $[^{82.9}_{70.1}]$	83.2 ms $[^{92.9}_{78.4}]$	85.8 ms $[^{88.8}_{80.0}]$
	ULFM	Emulated	80.0 ms $[^{86.2}_{75.5}]$	89.6 ms $[^{99.3}_{79.9}]$	84.3 ms $[^{87.9}_{77.8}]$

The second part of the table shows the performance of the `Team` collectives using the X10 runtime over the three MPI implementations. Comparing the bare MPI performance to the performance of X10 `Team` collectives in native mode shows high performance overhead and variability introduced by the X10 runtime. The overhead ranges between 0% to 12826% considering the three MPI implementations. It is expected for managed runtime systems like X10 to pay additional performance cost to raise the level of programming abstraction. Our experiment shows that the management activities of X10, which include task scheduling, garbage collection, handling termination detection, data serialization, and progressing immediate tasks, adds up to 10 ms to the performance of MPI collective operations. Due to the high variability of the performance of native `Team` operations, it is hard to draw conclusions from them on the resilience overhead of ULFM. For the native `Team` barrier collective, the median performance of ULFM without FT is slower than ULFM. However, the overlap between the variability ranges suggests a comparable performance between the two configurations. The same applies to the emulated allreduce collective.

The emulated `Team` collectives depend on MPI's point-to-point communication. The results show comparable performance using the three MPI implementations and no significant resilience overhead by ULFM. The results also show a significant performance advantage for native `Team` collectives over emulated collectives. With ULFM, the emulated collectives were 11×, 9×, 9× slower than native collectives for barrier, bcast, and allreduce, respectively.

From this experiment, we conclude that MPI-ULFM does not introduce a significant resilience overhead on failure-free executions. This makes it a promising transport layer for resilient high-level languages.

3.5.4 Resilient X10 over MPI-ULFM versus TCP Sockets

Resilience was initially supported in X10 over the X10RT-Sockets layer only. In this experiment, we study the potential speedup for X10 applications by switching from using sockets to using MPI-ULFM on a cluster of virtual machines connected by an IP over Infiniband network in the *NECTAR* cloud. With X10RT-Sockets, the only available option for `Team` is to use the emulated implementation. Although it is possible to use either emulated or native collectives with X10RT-MPI, the previous experiment indicates that the proper option to use for achieving better performance is the native collectives. That is why Table 3.3 compares the results of using sockets with emulated collectives against ULFM with native collectives. The top two rows show the performance results using the default X10 runtime, and the bottom two rows show the results with the resilient X10 runtime.

The results show that the ULFM backend results in significant performance gains in both non-resilient and resilient modes. In non-resilient mode, the emulated barrier, broadcast, and allreduce over sockets are respectively 1.7×, 2.5×, and 2.5× slower than the corresponding native collectives over ULFM. The slowdown is more significant in resilient mode, where the emulated collectives in the same order are 14.0×, 23.3×, and 14.9× slower than the native collectives. The native collectives did not introduce any measurable resilience overhead, since they avoid creating remote X10 asyncs that require expensive tracking by the resilient finish protocol. On the other hand, the emulated collectives suffered high resilience overhead: 703% for barrier, 747% for broadcast, and 482% for allreduce.

This experiment demonstrates that RX10 applications can benefit by switching from the socket backend to the ULFM backend to avoid the high resilience overhead of X10's emulated collectives. That is in addition to the direct benefit of scaling applications to larger core counts by utilizing the ever-growing parallelism in supercomputers. The next experiment evaluates the scalability of different `Team` operations in a supercomputer environment.

Table 3.3: Performance of Team collectives with sockets and ULFM on NECTAR with 360 places.

Runtime	Transport	Team Impl.	barrier	bcast	allreduce
X10	ULFM	Native	13.4 ms $[^{018.1}_{010.5}]$	12.4 ms $[^{017.7}_{002.9}]$	18.4 ms $[^{022.2}_{014.7}]$
	Sockets	Emulated	22.7 ms $[^{028.4}_{013.6}]$	31.5 ms $[^{041.1}_{025.0}]$	46.3 ms $[^{057.8}_{036.2}]$
RX10	ULFM	Native	13.0 ms $[^{016.3}_{011.4}]$	11.5 ms $[^{015.8}_{004.0}]$	18.1 ms $[^{023.8}_{015.3}]$
	Sockets	Emulated	182.0 ms $[^{195.8}_{166.7}]$	267.1 ms $[^{287.1}_{243.1}]$	269.5 ms $[^{291.8}_{248.4}]$

3.5.5 Team Construction Performance

In this experiment, we evaluate the performance of creating a `Team` object in both emulated and native modes, with X10 and RX10. In the emulated mode, constructing a `Team` object is performed using a remote **`finish`** that sends an **`async`** to each place to create place-local data strucutres for holding the new `Team` state. The performance of this remote finish varies between X10 and RX10 due to the additional book-keeping actitivites of the resilient finish protocol.

In the native mode, constructing a `Team` object results in creating a corresponding MPI communicator using the blocking collective operation `MPI_Comm_Create`. An additional call to `MPI_Comm_Shrink` is used if the X10 runtime is running in resilient mode. Orchestrating the places to create a new `Team` collectively is done at the X10RT-MPI layer through low-level messages. As explained in Section 3.4.4, we avoid the overhead of the resilient finish protocol by avoiding broadcasting remote tasks while creating a native `Team` object.

Table 3.4 shows the bare performance of `MPI_Comm_Create` and `MPI_Comm_Shrink` over ULFM. The results show a favourable sub-linear scalability for both operations, and that the cost of calling the two operations is less than 2 ms with 1024 places. On the other hand, constructing a native `Team` object takes up to 150 ms with 1024 places. That is because of the internal communication required to line up the places for invoking the communicator collectives, in addition to other runtime management activities that can cause processing delays.

In non-resilient mode, the performance of the emulated `Team` is comparable to the performance of the native `Team` with up to 256 places[3]. With larger place counts, the native `Team` outperforms the emulated `Team`. In resilient mode, the native `Team` significantly outperforms the emulated `Team` — creating an emulated `Team` took 505.1 ms with 109% resilience overhead, compared to only 147.8 ms for creating a native `Team` with 1024 places.

In resilient X10 applications, failure of places mandates reconstructing the `Team` objects. The performance improvement achieved by using ULFM's fault-tolerant collectives is therefore significant for speeding up the recovery process of applications.

Table 3.4: Team construction performance on Raijin with ULFM.

	MPI_Comm_Create	MPI_Comm_Shrink	Native	Emulated	
Places			**X10/RX10**	**X10**	**RX10**
128	0.26 ms $[^{0.27}_{0.25}]$	0.38 ms $[^{0.39}_{0.37}]$	117.7 ms $[^{135.6}_{024.5}]$	64.5 ms $[^{68.2}_{60.1}]$	190.0 ms $[^{198.9}_{180.2}]$
256	0.31 ms $[^{0.34}_{0.31}]$	0.43 ms $[^{0.43}_{0.43}]$	131.4 ms $[^{135.6}_{122.4}]$	74.2 ms $[^{168.4}_{66.9}]$	338.9 ms $[^{339.0}_{330.0}]$
512	0.39 ms $[^{0.40}_{0.38}]$	0.51 ms $[^{0.53}_{0.51}]$	135.6 ms $[^{137.5}_{135.3}]$	218.4 ms $[^{225.5}_{209.3}]$	369.7 ms $[^{376.7}_{359.1}]$
1024	0.49 ms $[^{0.53}_{0.49}]$	0.73 ms $[^{0.75}_{0.73}]$	147.8 ms $[^{148.8}_{145.9}]$	241.5 ms $[^{248.1}_{232.6}]$	505.1 ms $[^{527.5}_{496.8}]$

[3]The emulated mode was faster than the native mode with 128 and 256 places. However, we observed large variability in the performance results at these points which suggests that this result is not significant.

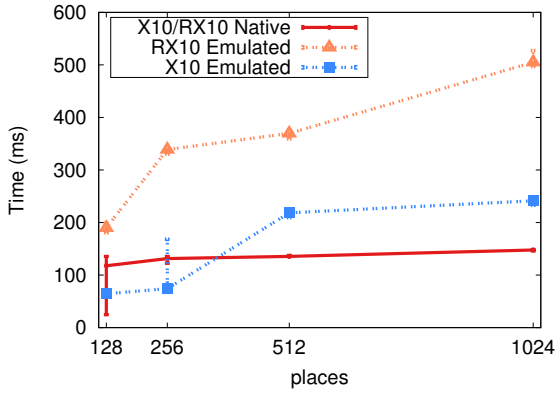

Figure 3.6: Team construction performance.

3.5.6 Team Collectives Performance

In this section, we evaluate the scalability of four `Team` operations: `barrier`, `broadcast`, `allreduce`, and `agree`. Where possible, we measured the bare MPI performance, the emulated performance, and the native performance. For the emulated collectives we report the performance using X10 and RX10. For the native collectives we report the same value for both X10 and RX10 since they are not subject to the resilience overhead of the **`finish`** termination detection protocol.

The `agree` interface uses the fault-tolerant agreement implementation provided by ULFM. We did not implement a corresponding emulated implementation for the `agree` interface due to time limitation. That is why we only report the bare MPI performance and the native `Team` performance of the `agree` interface.

The results are shown in Tables 3.5-3.8. Aligned with the results of the previous subsections, using the fault-tolerant MPI interfaces accelerates the performance of `Team` operations significantly compared to X10's emulated collectives. X10's bulk-synchronous applications can use optimized MPI collectives and execute resiliently with virtually no resilience overhead from the RX10 runtime.

Table 3.5: Team.Barrier performance on Raijin.

		Team.barrier()		
Places	MPI_Ibarrier/ MPI_Test	Native X10/RX10	Emulated X10	Emulated RX10
128	0.10 ms $[^{0.11}_{0.10}]$	1.7 ms $[^{15.0}_{00.3}]$	57.9 ms $[^{62.0}_{50.1}]$	61.9 ms $[^{071.4}_{057.6}]$
256	0.10 ms $[^{0.18}_{0.10}]$	6.1 ms $[^{16.1}_{00.5}]$	69.5 ms $[^{70.8}_{64.3}]$	189.9 ms $[^{198.6}_{175.1}]$
512	0.12 ms $[^{0.22}_{0.11}]$	7.7 ms $[^{15.3}_{02.0}]$	80.0 ms $[^{86.7}_{70.6}]$	212.9 ms $[^{221.2}_{201.1}]$
1024	0.17 ms $[^{0.23}_{0.11}]$	9.3 ms $[^{16.2}_{01.4}]$	88.1 ms $[^{98.7}_{83.0}]$	246.7 ms $[^{308.0}_{229.0}]$

Table 3.6: Broadcast performance on Raijin.

Places	MPI_Ibcast/ MPI_Test	Team.bcast() Native X10/RX10	Team.bcast() Emulated X10	Team.bcast() Emulated RX10
128	0.03 ms $[^{0.14}_{0.02}]$	7.6 ms $[^{15.0}_{00.4}]$	64.6 ms $[^{073.8}_{052.1}]$	100.0 ms $[^{110.0}_{090.8}]$
256	0.03 ms $[^{0.19}_{0.02}]$	9.7 ms $[^{13.9}_{01.7}]$	73.4 ms $[^{080.6}_{064.5}]$	120.0 ms $[^{130.0}_{110.5}]$
512	0.04 ms $[^{0.24}_{0.03}]$	10.6 ms $[^{15.1}_{02.2}]$	96.1 ms $[^{101.0}_{089.5}]$	270.0 ms $[^{278.9}_{257.4}]$
1024	0.04 ms $[^{0.11}_{0.03}]$	11.5 ms $[^{16.3}_{01.6}]$	103.8 ms $[^{109.9}_{095.4}]$	316.0 ms $[^{363.7}_{292.5}]$

Table 3.7: Allreduce performance on Raijin.

Places	MPI_Iallreduce/ MPI_Test	Team.allreduce() Native X10/RX10	Team.allreduce() Emulated X10	Team.allreduce() Emulated RX10
128	0.27 ms $[^{0.30}_{0.27}]$	4.4 ms $[^{16.1}_{00.4}]$	64.8 ms $[^{069.3}_{060.1}]$	110.2 ms $[^{117.8}_{102.0}]$
256	0.33 ms $[^{0.35}_{0.33}]$	5.8 ms $[^{14.8}_{01.4}]$	76.7 ms $[^{080.5}_{072.1}]$	130.1 ms $[^{134.0}_{129.0}]$
512	0.39 ms $[^{0.40}_{0.37}]$	9.4 ms $[^{16.0}_{01.4}]$	92.4 ms $[^{099.6}_{087.3}]$	269.4 ms $[^{274.3}_{258.7}]$
1024	0.45 ms $[^{0.47}_{0.44}]$	11.9 ms $[^{17.7}_{01.3}]$	101.3 ms $[^{109.6}_{095.5}]$	306.8 ms $[^{331.4}_{299.6}]$

Table 3.8: Agreement performance on Raijin.

Places	MPI_Comm_Iagree/ MPI_Test	Team.agree() Native X10/RX10	Team.agree() Emulated X10/RX10
128	0.11 ms $[^{0.11}_{0.11}]$	6.4 ms $[^{16.6}_{00.1}]$	NA
256	0.11 ms $[^{0.11}_{0.11}]$	7.1 ms $[^{15.7}_{01.2}]$	NA
512	0.12 ms $[^{0.13}_{0.12}]$	8.4 ms $[^{14.8}_{03.5}]$	NA
1024	0.12 ms $[^{0.13}_{0.12}]$	10.6 ms $[^{15.2}_{02.9}]$	NA

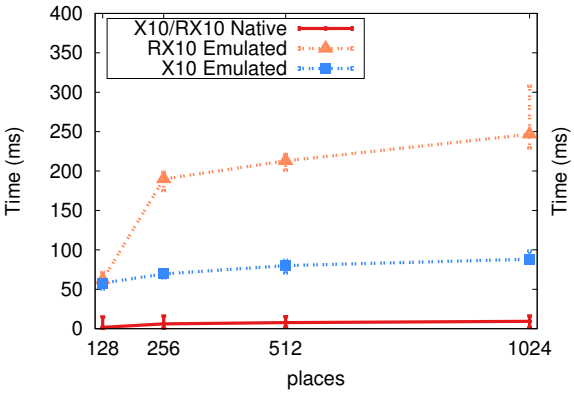

Figure 3.7: Team.Barrier performance.

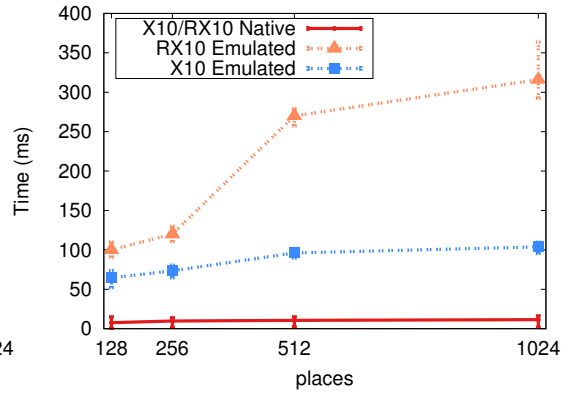

Figure 3.8: Team.Broadcast performance.

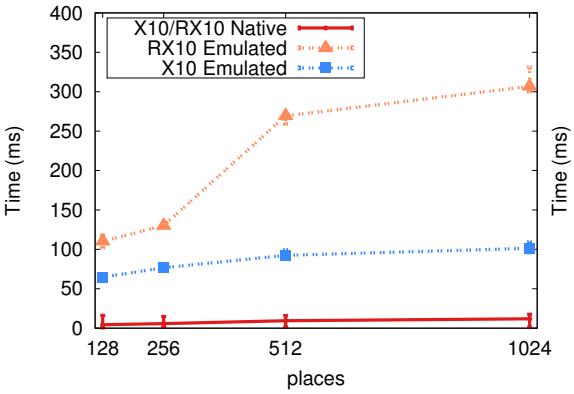

Figure 3.9: Team.Allreduce performance.

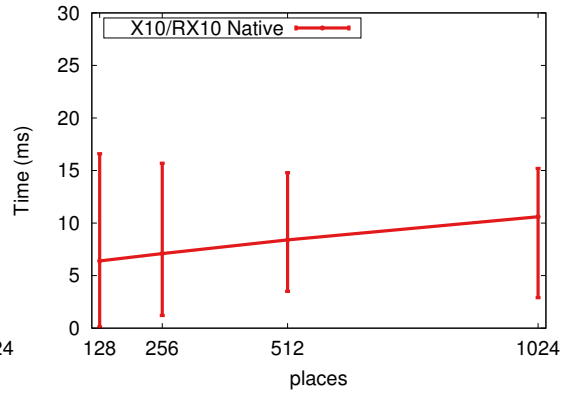

Figure 3.10: Team.Agree performance.

3.6 Related Work

To the best of our knowledge, our work is the first to explore the use of ULFM as a runtime system for high-level programming models. The closest to our work is related to the Fortran Coarrays programming model (Section 2.3.2.3). Fortran 2018 added new features to allow Coarrays programs to detect image failures and reform the remaining images for post-failure processing[4]. Fanfarillo et al. [2019] used ULFM to add these features to the OpenCoarrays framework [OpenCoarrays, 2019]. Failure recovery is performed collectively by the active processes using the revoke, shrink, and agreement functions of ULFM. In contrast, our work avoids global synchronization for recovery.

More explorations of ULFM have been done in the context of MPI applications and frameworks, which we review next.

Laguna et al. [2016] present an evaluation of ULFM's programmability in two common patterns of HPC applications: bulk-synchronous with static load balancing and master-worker with dynamic load balancing. They concluded that the absence of a non-shrinking communicator recovery interface in ULFM complicates the recovery procedures of statically balanced computations, since it requires updating the domain decomposition to map to an arbitrary number of ranks, which is complex for many applications and may even be infeasible. They also reported complexities in developing local recovery procedures for computations that rely on non-blocking computation, since ULFM does not report failures while initiating a non-blocking communication. Although detecting failures through return codes enables applications to determine the location of the failure in the code, they reported that this method results in significant changes and complications to the non-resilient code.

Gamell et al. [2016] used ULFM as a basis for comparing two in-memory checkpointing mechanisms implemented in the Fenix library. The first is neighbor-based — each process saves a copy of its own data and a copy of its neighbor's data. The second is checksum-based — each process saves a copy of its own data, and one process saves the checksum of the data of all the processes. In their experiments, neighbor-based checkpointing achieved better performance at the expense of higher memory footprint.

Teranishi and Heroux [2014] used ULFM to implement the Local Failure Local Recovery programming model, which aims to support the possible transition from global to local rollback recovery in certain HPC applications. The programming model provides simple abstractions for handling data and communicator recovery to enable MPI applications to adopt local non-shrinking recovery more easily. Three abstractions are provided: 1) a resilient communicator that uses spare processes to repair its internal structure, 2) a redundant storage interface that implements in-memory checksum-based checkpointing, and 3) an extendable programming template for building recoverable distributed data structures.

[4]Coarrays uses the name image to refer to a Fortran process.

Losada et al. [2017] used ULFM to deliver transparent non-shrinking recovery for MPI applications using the ComPiler for Portable Checkpointing (CPPC) [Rodríguez et al., 2010]. CPPC is a source-to-source compiler capable of transparently injecting checkpointing instructions in MPI programs at safe points discovered using static code analysis and variable liveness analysis. The above paper improved the generated code from CPPC to automatically handle failure detection, global failure propagation using `MPI_COMM_REVOKE`, communicator recovery using `MPI_COMM_SHRINK` and `MPI_COMM_SPAWN`, and computation restart.

In addition to the checkpoint/restart frameworks, other research groups have used MPI-ULFM for implementing algorithmic-based fault-tolerant applications.

Rizzi et al. [2016] used ULFM to design an approximate PDE solver that can tolerate both hard and soft faults. Algorithmic reformations were performed to the PDE to introduce filters for corrupt data and to add more task-parallelism that can fit a master-worker execution model. They favoured the master-worker model as it does require global communicator repair it the event of a worker failure. Depending on the computation stage, the described algorithm handles failure of a worker either by ignoring its results or by resubmitting its work to another worker. As in our work, the master detects the failure of any of the workers by continuously probing the communicator using `MPI_ANY_SOURCE`. The master is assumed to be fully reliable.

Ali et al. [2015] applied an algorithmic fault-tolerant adaptation of the sparse grid combination technique (SGCT) [Harding and Hegland, 2013] in the GENE (Gyrokinetic Electromagnetic Numerical Experiment) plasma application. Through data redundancy, introduced by the adapted SGCT, their implementation is capable of recovering lost sub-grids from existing ones more efficiently than using checkpoint/restart. Synchronous non-shrinking recovery of the MPI communicator is facilitated using ULFM.

3.7 Summary

This chapter has focused on the MPI backend of the X10 language, and how we extended it to support fault-tolerant execution using MPI-ULFM. We outlined the fault tolerance features that RX10 requires from the underlying communication layer to meet the requirements of the resilient async-finish model. We described the details of the integration between X10 and MPI-ULFM, including the use of MPI-ULFM fault-tolerant collectives by X10's `Team` collectives.

The experimental evaluation demonstrated strong performance and scalability advantages to RX10 applications by switching from the sockets backend to the MPI-ULFM backend. Most importantly, the performance of native `Team` collectives showed that RX10 bulk-synchronous applications can avoid most of the resilience overhead of the RX10 runtime system by using MPI-ULFM collectives. We will present a performance evaluation of a suite of resilient bulk-synchronous applications in Section 5.4.4.

In the next chapter, we describe another performance enhancement to RX10 by providing an optimistic termination detection protocol for resilient **`finish`**.

Chapter 4

An Optimistic Protocol for Resilient Finish

This chapter describes how we reduced the resilience overhead of RX10 applications by designing a message-optimal resilient termination detection (TD) protocol for the async-finish task model. We refer to this protocol as 'optimistic finish' since it favors the performance of failure-free executions over the performance of failure recovery. In this chapter, we evaluate the performance of this protocol using microbenchmarks of X10 programs representing different task graphs that are common in resilient X10 programs. In the next chapter, we evaluate the protocol using realistic X10 applications (see Section 5.4).

After an introduction in Section 4.1, we give an overview about different nested task parallelism models in Section 4.2, and different approaches to resilient TD in Section 4.3. After that, we prove the optimality limits of non-resilient and resilient async-finish TD protocols in Section 4.4 and highlight the challenges of adding failure awareness to the async-finish model in Section 4.5. We then describe the data structures and APIs used by the X10 runtime system for control-flow tracking in Section 4.6. The following sections explain different async-finish TD protocols: non-resilient finish (Section 4.7), pessimistic resilient finish (Section 4.8), and our proposed optimistic resilient finish (Section 4.9). Section 4.10 describes two resilient stores that can be used for maintaining critical TD metadata and Section 4.11 concludes with a performance evaluation.

This chapter is based on our paper "Resilient Optimistic Termination Detection for the Async-Finish Model" [Hamouda and Milthorpe, 2019].

4.1 Introduction

Dynamic nested-parallel programming models present an attractive approach for exploiting the available parallelism in modern HPC systems. A dynamic computation evolves by generating asynchronous tasks that form an arbitrary directed acyclic graph. A key feature of such computations is TD — determining when all tasks

in a subgraph are complete. In an unreliable system, additional work is required to ensure that termination can be detected correctly in the presence of component failures. Task-based models for use in HPC must therefore support resilience through efficient fault-tolerant termination detection protocols.

The standard model of termination detection is the diffusing computation model [Dijkstra and Scholten, 1980]. In that model, the computation starts by activating a coordinator process that is responsible for signaling termination. The status of the other processes is initially idle. An idle process becomes active only 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 are passing to activate an idle process [Venkatesan, 1989].

Most termination detection algorithms for diffusing computations assign a parental responsibility to the intermediate tasks, requiring each intermediate task to signal its termination only after its successor tasks terminate [Dijkstra and Scholten, 1980; Lai and Wu, 1995; Lifflander et al., 2013]. 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 [Blumofe et al., 1995], where a task calling 'sync' waits for join signals from the tasks it directly spawned. It is also similar to the execution model of the `cobegin` and `coforall` constructs of the Chapel language [Chamberlain et al., 2007]. While having the intermediate tasks as implicit TD coordinators is favorable for balancing the traffic of TD among tasks, it may add unnecessary blocking at the intermediate tasks that is not needed for correctness. The async-finish model does not impose this restriction; a task can spawn other asynchronous tasks without waiting for their termination. Unlike Cilk's sync construct, the **`finish`** construct waits for join signals from a group of tasks that are spawned directly or transitively from it.

To the best of our knowledge, the first resilient termination detection protocol for the async-finish model was designed by [Cunningham et al., 2014] as part of the RX10 project. This protocol does not force the intermediate tasks to wait for the termination of their direct successors. It maintains a consistent view of the task graph across multiple processes, which can be used to reconstruct the computation's 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 chapter, we review the pessimistic finish protocol and demonstrate that the requirement for a consistent view results in a significant performance overhead for failure-free execution. We propose the 'optimistic finish' protocol, an alternative message-optimal protocol that relaxes the consistency requirement, resulting in a faster failure-free execution with a moderate increase in recovery cost.

4.2 Nested Task Parallelism Models

Computations that entail nested termination scopes are generally classified as fully-strict or terminally-strict [Blumofe and Leiserson, 1999] (Figure 4.1). Blumofe and Leiserson [1999] 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 [1]. 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. [2009] provide more details on X10's implementation of the async-finish model, contrasting it with Cilk's spawn-sync model.

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.

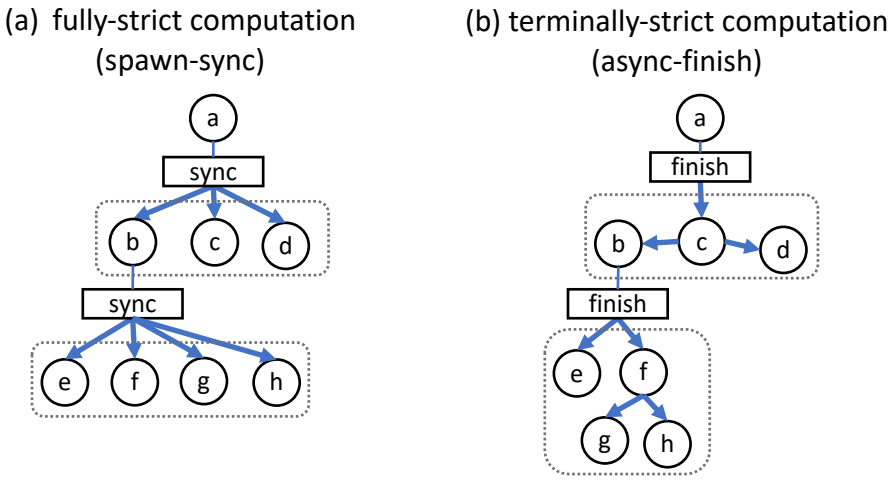

Figure 4.1: Nested parallelism models. A dotted-box represents a single termination scope. Circles represent tasks.

[1] Terminally-strict task graphs are also supported by the Chapel language using the sync statement.

4.3 Related Work

Termination detection is a well-studied problem in distributed computing, having multiple protocols proposed since the 1980s. Interested readers can refer to [Matocha and Camp, 1998] for a comprehensive survey of many TD algorithms.

The DS protocol, presented by Dijkstra and Scholten [1980], 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 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 [Lai and Wu, 1995; Lifflander et al., 2013].

Lai and Wu [1995] 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 [1989] 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 assumptions limit the algorithm's applicability to specialized systems.

Lifflander et al. [2013] 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 a k-node failure decreases as k increases [Meneses et al., 2012; Moody et al., 2010], 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. [2014]. They describe a protocol which separates the concerns of termination detection and survivability. They use a *finish resilient store* to maintain critical TD data for each `finish` construct. As the computation evolves, signals are sent to the resilient store to keep it as updated as possible with respect to 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 significant 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 [Lifflander et al., 2013] and [Cunningham et al., 2014] 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.

4.4 Resilient Async-Finish Optimality Limit

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 4.2, 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 Cunningham et al. [2014] 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 4.2-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

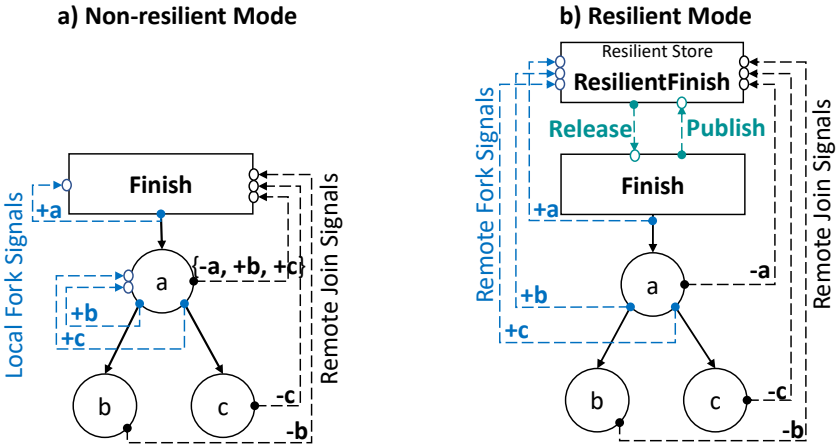

Figure 4.2: Message-optimal async-finish TD protocols.

(by decrementing `gc`). When the last task terminates, `gc` reaches zero, and the 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 4.2-a illustrates this method of non-resilient termination detection. We use $(+)$ as a `FORK` signal, and $(-)$ as a `JOIN` signal. When task `a` forks tasks `b` and `c`, it delays sending their `FORK` signals $(+\texttt{b}, +\texttt{c})$ until it joins. At this point, it packs its `JOIN` signal $(-\texttt{a})$ with the delayed `FORK` signals and sends one message containing the three signals $(-\texttt{a}, +\texttt{b}, +\texttt{c})$. 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 4.2-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 4.9) 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. □

4.5 Async-Finish Termination Detection Under Failure

4.5.1 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 that a message can be lost only if its source or destination has failed. We assume non-byzantine behavior. We do not assume any determinism in the order the network delivers messages.

4.5.2 Recovery Challenges

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 4.8 and Section 4.9, we describe how these challenges are addressed by the pessimistic protocol and the optimistic protocol, respectively.

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

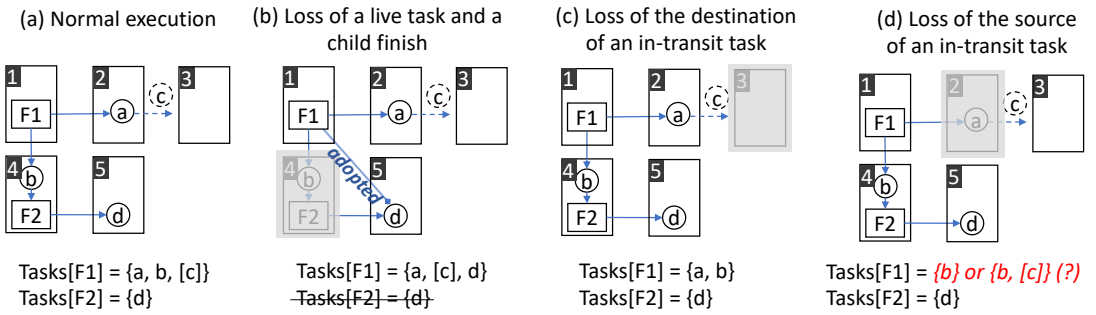

Figure 4.3: Task tracking under failure. The square brackets mark in-transit tasks. A dead place is covered by a gray rectangle.

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. Cunningham et al. [2014] 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 4.3-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 4.3-b results in losing the live task `b`, and the failure of place 3 in Figure 4.3-c results in losing the in-transit task `c`, because its target place is no longer available.

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 4.3-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.

4.6 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 X10's TD framework can be described using the pseudocode in Listing 4.1 and Figure 4.4. In Sections 4.7, 4.8, and 4.9, we will describe three termination detection protocols based on this framework.

Listing 4.1: 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 begin(src, dst):bool;
11    def join(src, dst):void;
12 }
```

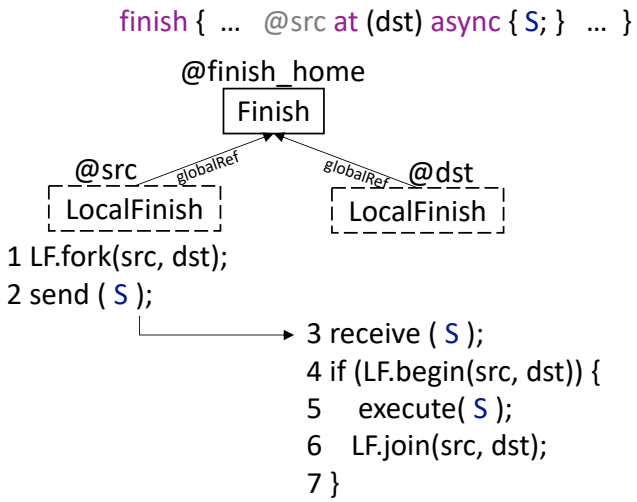

Figure 4.4: Tracking remote task creation.

4.6.1 Finish and LocalFinish Objects

A termination detection protocol is defined by providing concrete implementations of the abstract classes `Finish` and `LocalFinish` shown in Listing 4.1. 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 end of the finish block, after the finish creates all its direct tasks. When all tasks (direct and transitive) 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 so that the `Finish` has an up-to-date view of the global control structure.

4.6.2 Task Events

The abstract class `LocalFinish` defines three interfaces to track task events: `fork`, `begin`, and `join`. Figure 4.4 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. In a non-resilient TD protocol, `begin` may assume that all incoming tasks are valid. On the other hand, a resilient termination detection protocol may consider any task sent by a dead place as

invalid, even if the task is successfully received by the destination. Therefore, when failures are expected, `begin` may need to consult its governing `ResilientFinish` object to determine the validity of the received task.

In this chapter, 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.

4.7 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 4.2-a to use a maximum of one TD message per task.

Listing 4.2: Non-resilient finish pseudocode.

```
1 class NR_Finish(id) extends Finish {
2     gc:int=0; // global count
3     def merge(remoteLive) {
4         for (p in places) {
5            gc += remoteLive[p];
6         }
7         if (gc == 0)
8             release();
9     }
10 }
11 class NR_LocalFinish(id) extends LocalFinish {
12     lc:int=0; //local count
13     live:int[places]={0};
14     def fork(src, dst) {
15         live[dst]++;
16     }
17     def begin(src, dst) {
18         lc++;
19         return true;
20     }
21     def join(src, dst) {
22         live[dst]--;
23         lc--;
24         if (lc == 0)
25             @F[id].merge(live);
26     }
27 }
```

The pseudocode in Listing 4.2 captures 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 `LocalFinish` 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), `LocalFinish` passes its local `live` array to the finish object. Finish merges the passed array with its own `live` array and updates `gc` to reflect the current number of tasks. Termination is signaled when `gc` reaches zero.

4.7.1 Garbage Collection

The non-resilient finish implementation in X10 persists the `LocalFinish` objects in memory until the **`finish`** scope terminates. This enables the same object to be reused when tasks are repeatedly sent to the same place. When **`finish`** terminates, it sends asynchronous garbage collection messages to every place visited within the finish scope to delete the `LocalFinish` objects.

4.8 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 4.5. Using only the `FORK` and the `JOIN` signals (see Section 4.4), 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 `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 `live[d]` and `gc` (see Figure 4.5-a).

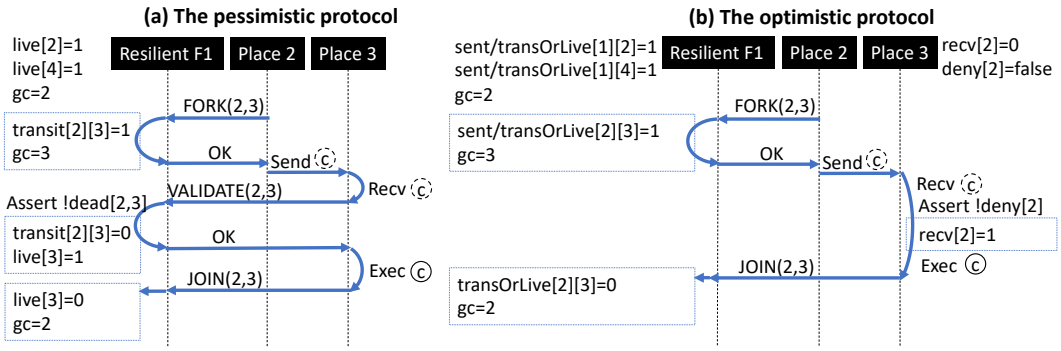

Figure 4.5: Task tracking events as task c transitions from place 2 to place 3, based on Figure 4.3-a.

4.8.1 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 4.4), 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 4.3 refers to this forwarding procedure.

4.8.2 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 4.3-Line 57). 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.

4.8.3 Garbage Collection

The pessimistic finish protocol recovers the control structure without collaboration with the `LocalFinish` objects. Therefore, it allows a `LocalFinish` object to be deleted automatically once its tasks terminate. Explicit garbage collection messages from a resilient finish to the `LocalFinish` objects are not required.

Listing 4.3: Pessimistic finish pseudocode.

```
1 abstract class P_ResilientStore {
2     def PUBLISH(id):void;
3     def ADD_CHILD(parentId, childId):void;
4 }
5 class P_Finish(id:Id) extends Finish {
6     def make(parent:Finish) {
7         @store.ADD_CHILD(parent.id, id);
8         @store.PUBLISH(id);
9     }
10 }
11 class P_LocalFinish(id:Id) extends LocalFinish {
12     def fork(src, dst) {
13         @RF[id].FORK(src, dst);
14     }
15     def join(src, dst){
16         @RF[id].JOIN(src, dst);
17     }
18     def begin(src, dst) {
19         return @RF[id].VALIDATE(src, dst);
20     }
21 }
22 class P_ResilientFinish(id:Id) {
23     gc:int=0;
24     live:int[places];
25     trans:int[places][places];
26     children:Set[Id];
27     adopter:Id;
28     def FORK(src, dst){
29         FORWARD_TO_ADOPTER;
30         if (bothAlive(src, dst)) {
31             trans[src][dst]++;
32             gc++;
33         }
34     }
35     def JOIN(src, dst) {
36         FORWARD_TO_ADOPTER;
37         if (!dst.isDead()) {
38             live[dst]--;
39             gc--;
40             if (gc == 0)
41                 @F[id].release();
42         }
43     }
44     def VALIDATE(src, dst) {
45         FORWARD_TO_ADOPTER;
46         if (bothAlive(src, dst)) {
47             trans[src][dst]--;
48             live[dst]++;
```

```
49              return true;
50          }
51          else
52              return false;
53      }
54      def addChild(cId) {
55          children.add(cId);
56      }
57      def recover(dead) {
58          // adopt orphaned tasks
59          for (c in children) {
60              if (c.home == dead) {
61                  trans += @RF[c].trans;
62                  live += @RF[c].live;
63                  gc += @RF[c].gc;
64                  @RF[c].adopter = id;
65              }
66          }
67          // exclude lost tasks
68          gc -= trans[dead][*] + trans[*][dead] + live[dead];
69          trans[dead][*] = 0;
70          trans[*][dead) = 0;
71          live[dead] = 0;
72
73          if (gc == 0)
74              @F[id].release();
75      }
76  }
```

4.9 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. The optimistic finish protocol makes the more practical assumption that a failure will impact a small minority of the system components [Lifflander et al., 2013; Sato et al., 2012; Meneses et al., 2012; Moody et al., 2010], and that the majority of components will be available and capable of collaborating to recover the control structure when failures occur.

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 4.4-Line 38). 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 4.4-Line 21). 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 4.5-b). Note that `sent[][]` and `recv[]` are never decremented. We will show in Section 4.9.2 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.

4.9.1 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 4.4). 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 58 in Listing 4.4). 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.

4.9.2 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 4.9.1), the resilient finish object can deduct `transOrLive[*][P]` from the global count and reset `transOrLive[*][P]` for each failed place `P`. Any `JOIN` messages received from the dead place `P` must be discarded, otherwise they may incorrectly alter the global count [2]. 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 4.4 and its call in Listing 4.4-Line 78).

4.9.3 Garbage Collection

The optimistic finish protocol requires the places visited within a `finish` scope to maintain their `LocalFinish` objects until finish terminates. That is because the `LocalFinish` objects collaborate with the resilient finish object during failure recovery as described above. When a resilient finish detects the termination of all its tasks, it sends explicit asynchronous garbage collection messages to all visited places to delete the `LocalFinish` objects.

[2] We do not assume any determinism in the order the network delivers messages. The cut of time for a resilient finish object to accept or ignore a JOIN message from a place is the time it detects the place's failure.

Listing 4.4: Optimistic finish pseudocode.

```
1 abstract class O_ResilientStore {
2     def PUBLISH(id, parentId):void;
3     def FIND_CHILDREN(id, place):Set[Id];
4 }
5 class O_Finish(id:Id) extends Finish {
6     def make(parent:Finish) {
7         @store.PUBLISH(id, parent.id);
8     }
9 }
10 class O_LocalFinish(id:Id) extends LocalFinish {
11     deny:bool[places];
12     recv:int[places];
13     def fork(src, dst) {
14         @RF[id].FORK(src, dst);
15     }
16     def join(src, dst){
17         @RF[id].JOIN(src, dst);
18     }
19     def begin(src, dst) {
20         if (deny[src]) {
21             return false;
22         } else {
23             recv[src]++;
24             return true;
25         }
26     }
27     def COUNT_TRANSIT(nSent, dead) {
28         deny[dead] = true;
29         return nSent - recv[dead];
30     }
31 }
32 class O_ResilientFinish(id:Id) {
33     gc:int=0;
34     parent:Id;
35     transOrLive:int[places][places];
36     sent:int[places][places];
37     ghosts:Set[Id]; isGhost:bool;
38     def FORK(src, dst){
39         if (bothAlive(src, dst)){
40             transOrLive[src][dst]++;
41             gc++;
42             sent[src][dst]++;
43         }
44     }
45     def JOIN(src, dst){
46         if (!dst.isDead()) {
47             transOrLive[src][dst]--;
48             gc--;
```

```
49              tryRelease();
50          }
51      }
52      def removeChild(ghostId) {
53          ghosts.remove(ghostId); tryRelease();
54      }
55      def tryRelease() {
56          if (gc == 0 && ghosts.empty()) {
57              if (isGhost)
58                  @RF[parent].removeChild(id);
59              else
60                  @F[id].release();
61          }
62      }
63      def recover(dead) {
64          // adopt orphaned tasks
65          if (transOrLive[*][dead] > 0) {
66              val c = @store.FIND_CHILDREN(id, dead);
67              ghosts.addAll(c);
68              for (g in c)
69                  @RF[g].isGhost = true;
70          }
71
72          // exclude lost tasks
73          gc -= transOrLive[*][dead];
74          transOrLive[*][dead] = 0;
75          for (p in places) {
76              if ( transOrLive[dead][p] > 0) {
77                  val s = sent[dead][p];
78                  val t = @LF[id].COUNT_TRANSIT(s, dead);
79                  transOrLive[dead][p] -= t;
80                  gc -= t;
81              }
82          }
83
84          tryRelease();
85      }
86  }
```

4.9.4 Optimistic Finish TLA Specification

Temporal Logic of Actions (TLA) [The TLA Home Page] is a language for specifying and automatically verifying 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. The TLC tool has a distributed implementation and a centralized implementation with different capabilities.

We developed a formal model for the optimistic finish protocol using TLA to verify the protocol's correctness. 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 c children. It can also simulate the occurrence of one or more place failures as the task graph evolves.

The full specification is included in Appendix B. It defines the protocol behaviour using the 22 actions listed in Table 4.1. We defined a guard on each action that determines the preconditions that must be satisfied before the action is activated. A correct execution occurs only if the guard of the action `ProgramTerminating` is eventually satisfied, which indicates the termination of the root finish of the graph.

Table 4.1: TLA+ actions describing the optimistic finish protocol.

Type	Actions list
Task actions	CreatingFinish
	CreatingRemoteTask
Finish actions	TerminatingTask
	ReceivingPublishDoneSignal
	ReceivingReleaseSignal
	CreatingRemoteTask
LocalFinish actions	TerminatingTask
	MarkingDeadPlace
	DroppingTask
Communication actions	SendingTask
	ReceivingTask
	ReceivingPublishSignal
	ReceivingTransitSignal
	ReceivingTerminateTaskSignal
	ReceivingTerminateGhostSignal
ResilientFinish actions	FindingGhostChildren
	AddingGhostChildren
	CancellingTasksToDeadPlace
	SendingCountTransitSignalToLocalFinish
	CancellingTransitTasksFromDeadPlace
Failure actions	KillingPlace
Program termination action	ProgramTerminating

The distributed TLC tool currently cannot validate liveness properties, such as 'the system must eventually terminate', which we needed to guarantee in our protocol. The centralized TLC tool can perform this validation; however, it is not scalable. Using the centralized 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 the 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. We used three places in our testing.

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 the correctness of our protocol over a 3-level task tree with a branching factor of 2 using 3 places, where a failure can impact one place at any point in the execution. Interested readers can check the specification and the outputs of verification in the public GitHub repository [X10 Formal Specifications].

Based on the challenges described in Section 4.5, we believe the used graph configuration is sufficient for detecting any errors in the protocol. The model will not reach the `ProgramTerminating` state if handling the loss of in-transit and live tasks is incorrect, because the root finish will wait indefinitely for tasks that has died. Similarly, if the adoption mechanism is incorrect, the root finish will wait indefinitely for the `REMOVE_CHILD` signal from its ghost child. The TLC tool verified that the `ProgramTerminating` state is eventually reached, which means that the root finish does not block indefinitely and is eventually released. This result demonstrates that the optimistic protocol can recover the control flow of a computation correctly.

4.10 Finish Resilient Store Implementations

The resilient store is a key factor in the performance and the survivability of the TD implementation. Cunningham et al. [2014] used three resilient store types to evaluate the overhead of RX10 using the pessimistic finish protocol. One of the stores is based on an off-the-shelf Java-based resilient store called ZooKeeper [Hadoop/ZooKeeper], which is not suitable for HPC platforms. The two other stores are writen in X10 itself, therefore, they can execute on HPC platforms by compiling them to Native X10 programs over MPI-ULFM [3].

One of the two HPC-suitable stores is a centralized store that keeps all resilient finish objects at place-zero assuming that place-zero will survive all failures. The centralized nature of the store simplifies the implementation of the resilient protocol. However, it results in a performance bottleneck with large numbers of concurrent

[3] Cunningham et al. [2014] evaluated the three stores over X10RT-Sockets, which was the only transport layer with resilience support at that time.

finish objects and tasks. The other store is a distributed store that replicates the state of 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. Each task and finish signal is synchronously replicated on both replicas. When the master replica fails, the backup replica detects the failure and triggers the adoption mechanism defined by the protocol. Unfortunately, this implementation was later removed from the code base of RX10 due to its complexity and instability. As a result, users of RX10 are currently limited to using the non-scalable centralized place-zero finish store.

4.10.1 Reviving the Distributed Finish Store

Because a centralized place-zero finish store can significantly limit the performance of RX10, we decided to reimplement a distributed finish store for RX10 for both optimistic and pessimistic protocols. Using TLA's model checker, we identified a serious bug in the replication protocol described in [Cunningham et al., 2014] 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 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 RX10 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 (the same optimization is used in [Cunningham et al., 2014]). The full specification of the replication protocol is available in Appendix C.

In the performance evaluation section, we use these acronyms to refer to the different implementations of the optimistic and pessimistic protocols: pessimistic place-zero finish (P-p0), optimistic place-zero finish (O-p0), pessimistic distributed finish (P-dist), and optimistic distributed finish (O-dist).

4.11 Performance Evaluation

In this chapter, we evaluate the optimistic and pessimistic finish protocols using micro-benchmarks that represent patterns of task graphs with different communication intensities. We compare the centralized and distributed implementations of these

protocols, aiming to identify the protocol-implementation combination that delivers best performance for certain task graphs. In the next chapter, in Section 5.4.4, we perform the same evaluation using representative applications from the molecular dynamics and machine learning domains.

4.11.1 Experimental Setup

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 and statically bound each place to a separate core. Each core is oversubscribed with two internal place threads: a worker thread[4] for handling user tasks, and an immediate thread[5] for handling low-level runtime communication.

We use one core per place because most of the evaluated distributed task patterns create one task at the majority of places (i.e. single remote task, flat fan-out, flat fan-out message back, tree fan-out, and ring around). Only two distributed patterns (the all-to-all patterns) create large number of local parallel tasks per place. For consistency, we decided to evaluate all the patterns using the same mapping configuration. We believe that one mapping strategy is sufficient for the purpose of the experiments, which is comparing the overhead of the different TD protocols.

We built MPI-ULFM from revision `e87f595` of the `master` branch of the repository at `https://bitbucket.org/icldistcomp/ulfm2.git`. We built the X10 compiler and runtime 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.

4.11.2 BenchMicro

We use the `BenchMicro.x10` program, designed by Cunningham et al. [2014] for evaluating the resilience overhead of X10 in representative computation patterns[6]. We modified the program 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 also modified the warm-up procedure to only execute the all-to-all pattern, rather than executing all the patterns, to save execution time on *Raijin*. The all-to-all pattern puts each place in communication with each other place, which we find sufficient for warming up the transport layer at all places.

[4]One worker thread is configured by setting the environment variable X10_NTHREADS=1

[5]One immediate thread is configured by setting the environment variable X10_NUM_IMMEDIATE_THREADS=1

[6]The source code of BenchMicro is available at x10/x10.dist/samples/resiliency/BenchMicro.x10

For each computation pattern, we measure its execution time with the following runtime configurations:

- non-resilient: X10 with the default non-resilient finish implementation available in the X10 2.6.1 release.
- P-p0: RX10 using the place-zero pessimistic finish available in the X10 2.6.1 release.
- O-p0: RX10 using our proposed place-zero optimistic finish.
- P-dist: RX10 using our proposed distributed pessimistic finish.
- O-dist: RX10 using our proposed distributed optimistic finish.

We measured the performance using 256, 512 and 1024 places, with one place per core. Each pattern in each configuration was executed 30 times. In the following figures, we show the median with error bars representing the range between the 25th percentile, and the 75th percentile. Table 4.2 and Table 4.3 summarize the performance results with 1024 places.

4.11.2.1 Performance Factors

To clarify the performance results obtained by the different finish implementations, we review two important performance factors:

Garbage Collection Messages: The pessimistic finish implementations do not maintain remote references to the `LocalFinish` objects created at places visited within a finish scope. The local garbage collector of each place eventually deletes these objects after their tasks complete. On the other hand, the non-resilient finish and the optimistic resilient finish implementations maintain remote references to the local finish objects for future use. After a finish terminates, explicit garbage collection messages are sent asynchronously to clean up the `LocalFinish` objects. The absence of Garbage Collection (GC) messages in the pessimistic implementations gives them a performance advantage over the optimistic implementations. The GC messages can also be a source of performance variability for the optimistic finish implementations. In the `BenchMicro` program for example, running multiple iterations of the same pattern to obtain aggregate performance results is likely to cause GC messages from a previous iteration to interfere with TD messages of a current iteration.

Releasing a Finish: Depending on the finish implementation, releasing a finish can be done locally or through a remote communication. The non-resilient implementation and the resilient distributed implementations release a finish locally, whereas the resilient place-zero implementations release a finish remotely. The non-resilient finish can directly release itself because it handles its termination detection independently. A resilient place-zero finish delegates termination detection to the resilient finish object at place zero; therefore, releasing a finish requires a remote message from place zero to the finish home place. On the other hand, a resilient distributed finish delegates termination detection to the master replica located at the same place as the

finish; therefore, releasing a finish is done locally. To conclude, as the number of concurrent finishes increases, the cost of releasing a finish is expected to grow linearly with the place-zero implementations and to remain constant with the distributed implementations.

4.11.2.2 Performance Results

In the following sections, we analyse the scalability and the resilience overhead of different computation patterns.

1. Local Finish

The first pattern represents a local finish governing 100 local activities, as performed by this code:

```
1 finish { for (i in 1..100) async S; }
```

Spawning local parallel tasks is commonly used in X10 programs for exploiting multi-core parallelism within a node. The first release of RX10, in version 2.4.1, did not differentiate tasks spawned locally from tasks spawned remotely; all tasks involved interactions with the resilient store for tracking their state. As a result, a local finish governing 100 tasks could suffer a slowdown of up to 1000x with a centralized resilient finish implementation [Grove et al., 2019].

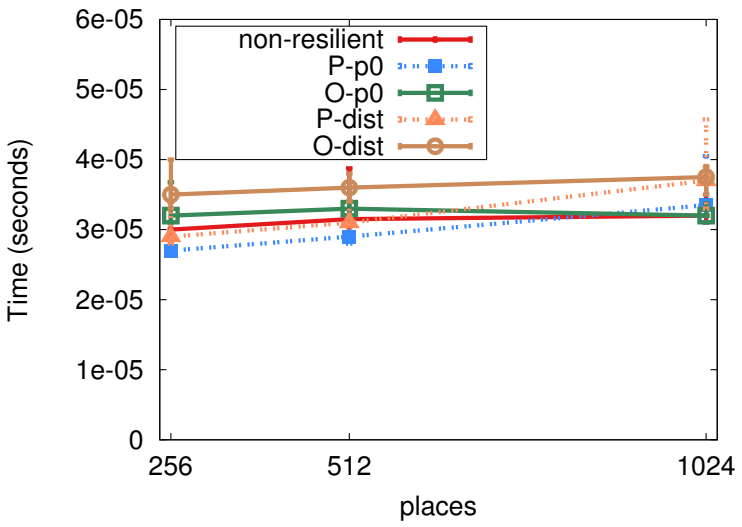

Figure 4.6: BenchMicro: local finish performance.

A major enhancement to RX10, in version 2.6.1, was achieved by the X10 team by omitting interactions with the resilient store for tasks spawned locally [Grove et al., 2019]. Since the failure of a place results in the immediate loss of its local tasks, using the resilient store to track the exact number of local tasks at each place is not useful. Hence, a non-resilient local counter is sufficient for tracking a local fragment of the

task tree at a certain place. When the counter reaches zero, the place sends a single message to the resilient store to report the termination of the root task of the local fragment, only if the root task was received from another place. Following the same concept, a finish does not need to be recorded in the resilient store until it spawns a remote task. Consequently, the resilience overhead of a local finish can be virtually eliminated regardless of how many local tasks it controls.

All four resilient finish implementations under evaluation in this chapter include the local task and finish optimizations. The results in Figure 4.6 show comparable performance between the different finish configurations. Minor implementation differences are behind the slight variation in performance. As shown in Table 4.2, with 1024 places, there is no significant slowdown for the place-zero finish implementations versus non-resilient finish. The distributed implementations are slower by a factor of 1.2 not due to a communication overhead, but due to a minor implementation difference in the way the finish objects are created and recorded locally.

2. Single Remote Task

The second pattern is for a finish that spawns a single asynchronous remote task at the next place, as performed by the following code:

```
1 val home = here;
2 val next = Place.places().next(home);
3 finish at (next) async S;
```

This pattern creates a `Finish` object at the home place and a `LocalFinish` object at the next place. Despite the additional asynchronous GC message for deleting the `LocalFinish` object, optimistic finish outperforms pessimistic finish in this pattern, as shown in Figure 4.7. With 1024 places, the performance improvement achieved by the optimistic protocol is 18% with a place-zero finish and 29% with a distributed finish.

Because the number of spawned tasks does not depend on the number of places, this pattern is expected to achieve a constant scaling. However, the results show an unexpected increase in the execution time as the number of places increases, even in non-resilient mode. The same trend was observed when we assigned two cores per place on *Raijin* to reduce the possible jitter due to oversubscribing each core with two threads, and when testing this pattern using native X10 over sockets on the *NECTAR* cloud. In contrast, the expected non-increasing performance was achieved using the managed X10 runtime (i.e. X10 compiled to Java) over sockets on the *NECTAR* cloud. Therefore, we conjecture that there is a limitation in the native X10 runtime that causes it to suffer a slight decrease in performance as the number of places increases, and that the increasing execution time in the "single remote task" pattern is not due to the implementation of finish.

3. Flat Fan-Out

A typical X10 program uses a flat fan-out at least once during execution to distribute the work over the available places. In a flat fan-out, a finish spawns an asynchronous

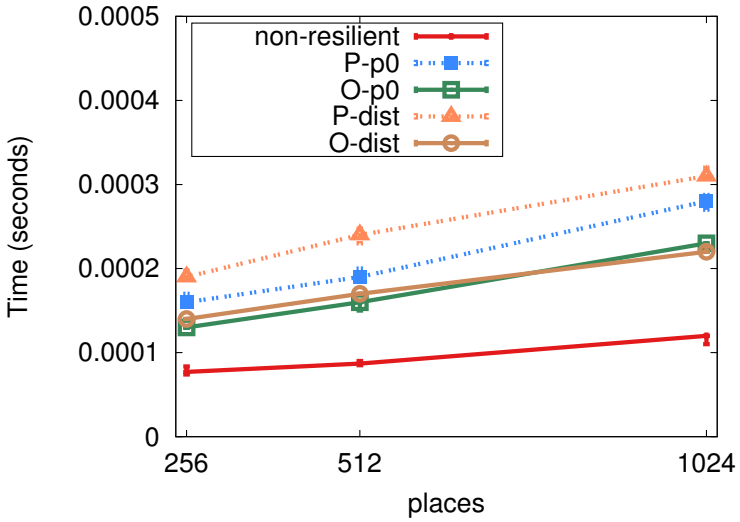

Figure 4.7: BenchMicro: single remote task performance.

task directly at each place, as shown in the following code:

```
1 finish { for (p in places) at (p) async S; }
```

This pattern is expected to achieve linear scalability as the number of places increases. Figure 4.8 shows that the different finish implementations indeed scale linearly, and that the optimistic finish outperforms pessimistic finish in most cases. With 1024 places, the performance improvement achieved by the optimistic protocol is 59% with a place-zero finish and only 14% with a distributed finish. We conjecture that the high performance variability with P-p0 with 1024 places is due to the interference of the immediate thread with the worker thread at place zero.

Assuming the number of places is N, the expected total number of messages for tracking the tasks is: $3 * N$ for P-p0 (N to fork, N to begin, and N to terminate) and $2 * N$ for O-p0 (N to fork and N to terminate). In a distributed finish, the master replica is hosted at the home place of the finish. Thus tasks originating from the finish home execute the fork signal at the master replica locally. The fork signals of the backup replica and all the begin and terminate signals generate remote messages. Therefore, the total number of TD messages for a flat fan-out is $5 * N$ for P-dist (N to fork, $2N$ to begin, and $2N$ to terminate) and $3 * N$ for O-dist (N to fork and $2N$ to terminate). The increase in number of TD messages and the use of synchronous replication causes a distributed finish to suffer significant resilience overhead in this pattern. Moreover, since this pattern creates only one finish, the scalability advantage of a distributed finish store is not leveraged.

X10 applications usually contain multiple task patterns, of which flat fan-out is only one. The cost of the fan-out pattern in the distributed modes may result in high performance penalty for applications that can benefit from the scalability of a distributed finish in other patterns. One possible optimization for a distributed

finish is to avoid replicating the finishes whose home is place zero [Cunningham et al., 2014]. While in BenchMicro we start all the patterns from the middle place, in practice, most (if not all) X10 applications start a flat fan-out from place zero. Because the failure of place zero is considered catastrophic in RX10, resilient termination detection for a finish hosted at place zero is simply wasted work. This optimization is available implicitly in the resilient place-zero finish implementations, because a finish belonging to place zero is not replicated elsewhere. Therefore, it is fair to add this optimization to the distributed finish implementations as well. We used this optimization to achieve better performance with distributed finish for some applications that we evaluate in the next chapter.

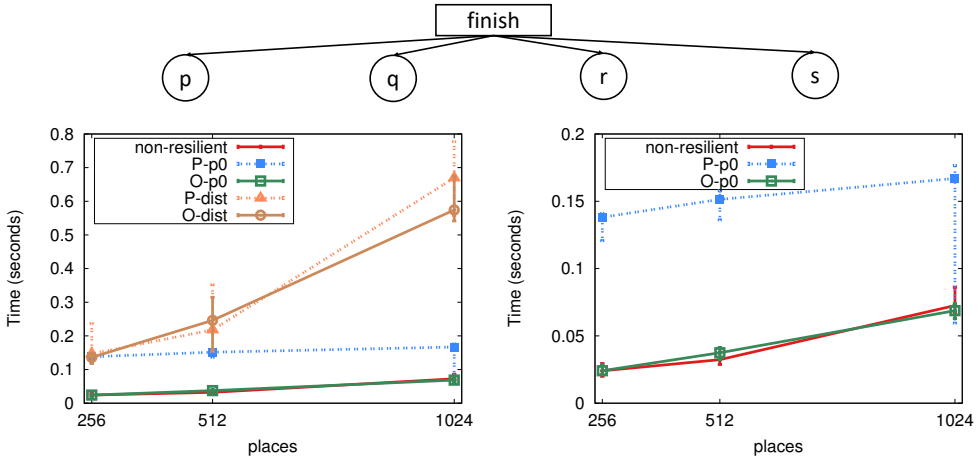

Figure 4.8: BenchMicro: flat fan-out performance. The left figure shows the scaling pefor-mance for all finish modes; The right figure zooms in on the same results for place-zero modes only.

4. Flat Fan-Out Message Back

In this pattern, a finish governs a flat fan-out to all places, and each place responds back by spawning an asynchronous message at the finish home, as performed by this code:

```
1 val home = here;
2 finish {
3     for (p in places) at (p) async {
4         at (home) async S;
5     }
6 }
```

This pattern is often used for gathering partial results computed by individual places at one place. Its scaling performance, as shown in Figure 4.9, is generally similar to the scaling performance of the fan-out pattern in Figure 4.8. However, doubling the number of tasks governed by the finish makes the pessimistic resilience

overhead more evident in this pattern than in the fan-out pattern, especially with a distributed finish. The performance improvement achieved by the optimistic protocol is 15% with a place-zero finish and 68% with a distributed finish. We conjecture that the high performance variability with O-p0 with 512 and 1024 places is due to interference from garbage collection messages and/or the immediate thread.

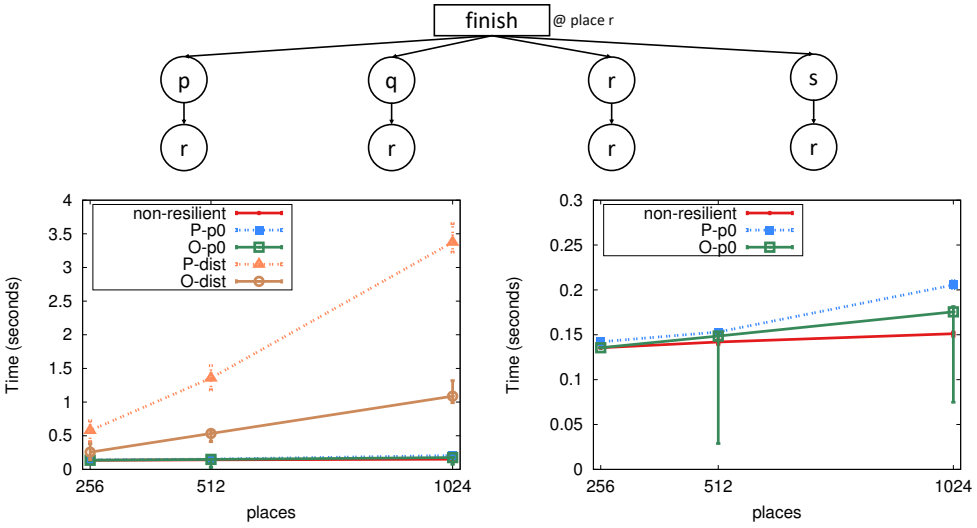

Figure 4.9: BenchMicro: flat fan-out message back performance. The left figure shows the scaling peformance for all finish modes; The right figure zooms in on the same results for place-zero modes only.

5. Tree Fan-Out

The tree fan-out pattern traverses the places recursively assuming a binary tree topology of the places. The root place creates the top most finish which spawns two asynchronous tasks at its direct children. The child receives the task and creates a nested finish that spawns two other tasks at its children, and so on. The following code demonstrates this pattern:

```
1 def traverse() {
2    if (noChildren()) return;
3    finish {
4       at (getChild1(here)) async { traverse(); }
5       at (getChild2(here)) async { traverse(); }
6    }
7 }
```

With only three points, the performance results in Figure 4.10 seem to show an almost constant scaling; however, this pattern is expected to achieve a logarithmic scaling as the number of places increases. The tree fan-out pattern creates a **finish** at each intermediate place, governing two tasks only. Because the task-to-finish ratio is small, the relative advantage of the optimistic finish is not significant. Meanwhile, the

results show a strong advantage for the distributed finish implementations compared to the centralized implementations. The large number of concurrent finishes created by this pattern generates a sequential bottleneck at place zero, which significantly slows down the execution. On the other hand, the distributed finish achieves better performance by balancing the termination detection load among the places. With 1024 places, the slowdown factor versus non-resilient finish is 8.6, 8.5, 1.4, 1.1 for P-p0, O-p0, P-dist, O-dist, respectively. The performance improvement achieved by the optimistic protocol is 2% with a place-zero finish and 27% with a distributed finish.

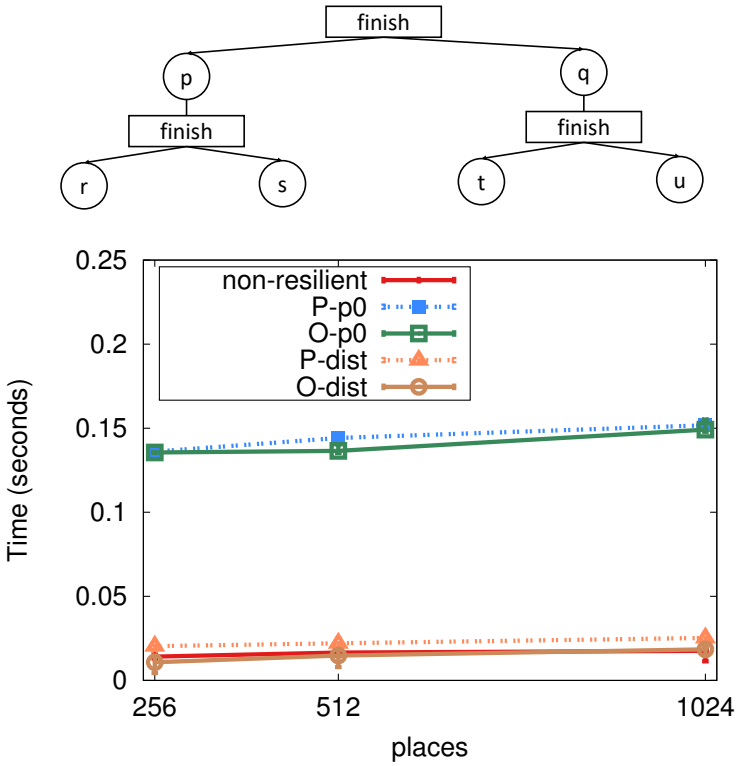

Figure 4.10: BenchMicro: tree fan-out performance.

6. All-To-All

The all-to-all pattern creates a single **finish** that governs a fan-out to all the places, followed by a fan-out from each place to all the places. The following code demonstrates this process:

```
1 finish {
2     for (p in places) at (p) async {
3         for (q in places) at (q) async S;
4     }
5 }
```

A graphical demonstration for this pattern using four places is in Figure 4.11. If the number of places is N, a single **finish** will track a total of $N^2 + N$ tasks directly. Reducing the task-level TD signals by adopting the optimistic finish protocol results in significant performance improvement for this pattern. With 1024 places, the performance improvement achieved by the optimistic protocol is 53% with a place-zero finish and 59% with a distributed finish.

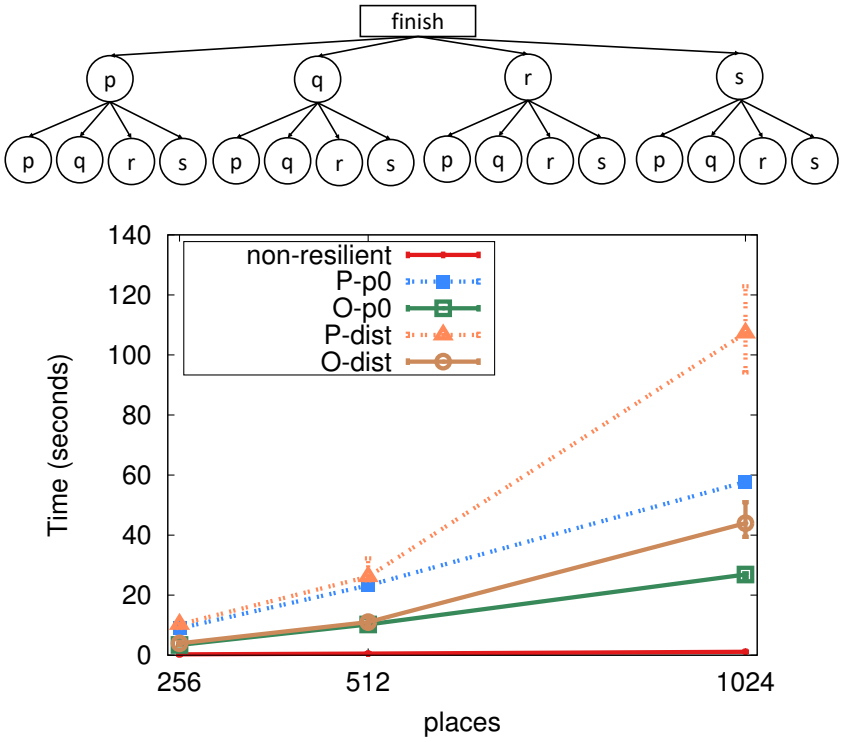

Figure 4.11: BenchMicro: all-to-all performance.

7. All-To-All With Nested Finish

This pattern is similar to the all-to-all pattern with the exception that each internal fan-out is governed by its own finish. As shown in the code below, the top finish in the first line governs N nested finishes, one per place.

```
1 finish {
2     for (p in places) at (p) async {
3         finish {
4             for (q in places) at (q) async S;
5         }
6     }
7 }
```

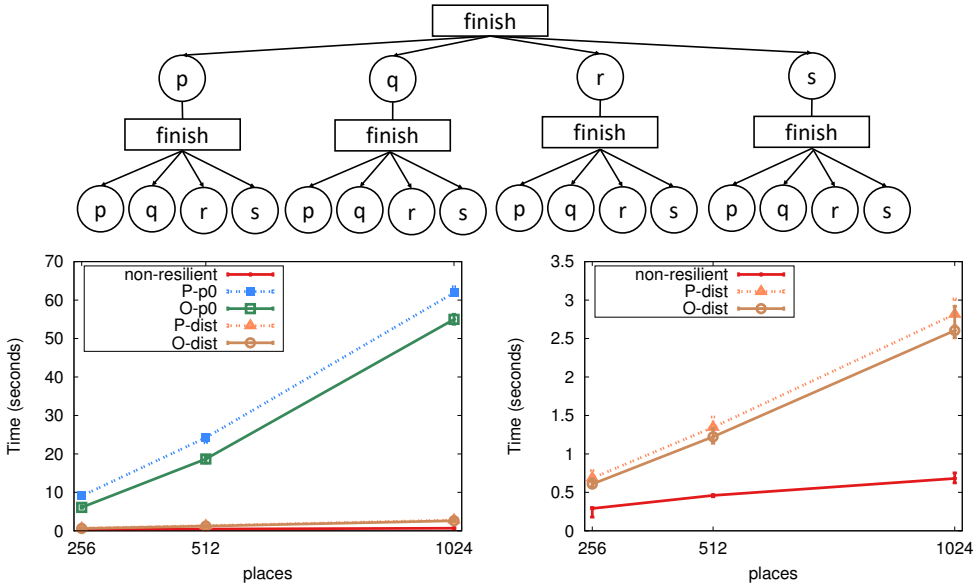

Figure 4.12: BenchMicro: all-to-all with nested finish performance. The left figure shows the scaling peformance for all finish modes; The right figure zooms in on the same results for distributed modes only.

Decomposing a large finish to multiple concurrent finishes at different places can improve the performance by allowing the TD logic to be executed in parallel at different places. Our results show that adding the nested finish to the all-to-all pattern improves the performance by 40% in non-resilient mode. A more significant improvement is achieved in resilient mode using the distributed finish implementations: 97% with P-dist and 94% with O-dist. In contrast, the centralized implementations, P-p0 and O-p0, are respectively 7% and 105% slower due to the sequential bottleneck imposed by place-zero and the added cost of creating and releasing the concurrent finish objects.

Modest improvement is credited to the optimistic finish protocol in this pattern. With 1024 places, the performance improvement due to the optimistic protocol is 11% with a place-zero finish and 7% with a distributed finish. The "all-to-all with nested finish" pattern basically executes N parallel "flat fan-outs". To put the numbers in context, remember that the improvement in the flat fan-out pattern with 1024 places is 59% with a place-zero finish and 14% with a distributed finish. The centralized nature of the place-zero implementations prevents the "all-to-all with nested finish" pattern from exploiting the available distributed parallelism and results in less performance efficiency. On the other hand, the improvement achieved with a distributed finish is comparable to the "flat fan-out" pattern.

8. Ring Around Via At

Assuming a ring topology for the places, this pattern traverses the places in a clockwise direction using the **at** construct, as shown below:

```
1 def ring(destination:Place):void {
2     if (destination == here) return;
3     val next = Place.places().next(here);
4     at (next) ring(destination);
5 }
```

For N places, this pattern creates N tasks, one per place, all chained together such that a left task is pending on the termination of the right task. Blocking is achieved by the **at** construct, which implicitly creates a **finish** to track the task executing the body of the **at**. The choice of the resilient finish implementation has little effect in this pattern due to sequential nature of the execution and the small task-to-finish ratio. Figure 4.13 shows that the four resilient implementations achieve almost the same performance. With 1024 places, the slowdown factor versus non-resilient finish is 1.8, 1.7, 1.8, 1.8 for P-p0, O-p0, P-dist, O-dist, respectively.

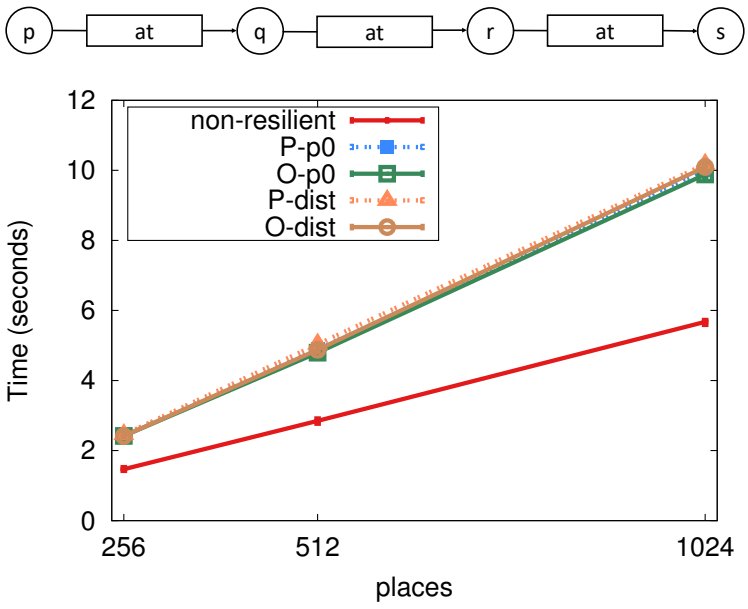

Figure 4.13: BenchMicro: ring around via at performance.

4.11.3 Conclusions

From the previous analysis, we can draw the following conclusions:

1. Our proposed optimistic protocol is successfully reducing the resilience overhead of RX10. Implementing more efficient GC mechanisms for deleting remote `LocalFinish` objects may further improve the performance of the optimistic finish implementations.

2. The effect of the optimistic protocol is more evident 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 better the performance of the resilient distributed finish implementations.

Table 4.2: Execution time in seconds for BenchMicro patterns with 1024 places.

	Pattern	Non-resilient	P-p0	O-p0	P-dist	O-dist
1	Local finish	3.20E-05	3.35E-05	3.20E-05	3.70E-05	3.75E-05
2	Single remote task	1.20E-04	2.80E-04	2.30E-04	3.10E-04	2.20E-04
3	Flat fan-out	7.25E-02	1.67E-01	6.87E-02	6.69E-01	5.74E-01
4	Flat fan-out message back	1.51E-01	2.05E-01	1.76E-01	3.37E+00	1.09E+00
5	Tree fan-out	1.76E-02	1.52E-01	1.49E-01	2.52E-02	1.85E-02
6	All-to-all	1.12E+00	5.77E+01	2.69E+01	1.07E+02	4.40E+01
7	All-to-all with nested finish	6.81E-01	6.19E+01	5.50E+01	2.81E+00	2.60E+00
8	Ring around via at	5.67E+00	1.00E+01	9.88E+00	1.02E+01	1.01E+01

Table 4.3: 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 accredited to the optimistic finish protocol.

	Pattern	P-p0	O-p0	Opt. %	P-dist	O-dist	Opt. %
1	Local finish	1.0	1.0	**0%**	1.2	1.2	**0%**
2	Single remote task	2.3	1.9	**18%**	2.6	1.8	**29%**
3	Flat fan-out	2.3	0.9	**59%**	9.2	7.9	**14%**
4	Flat fan-out message back	1.4	1.2	**15%**	22.3	7.2	**68%**
5	Tree fan-out	8.6	8.5	**2%**	1.4	1.1	**27%**
6	All-to-all	51.4	23.9	**53%**	95.6	39.2	**59%**
7	All-to-all with nested finish	90.9	80.8	**11%**	4.1	3.8	**7%**
8	Ring around via at	1.8	1.7	**1%**	1.8	1.8	**0%**

4.12 Summary

This chapter presented *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 reduces the resilience overhead of communication-intensive and highly-decomposed task-parallel kernels. Our `BenchMicro` analysis, which evaluates the resilience overhead of different computation patterns, gives insights into the performance of RX10 with different TD implementations and encourages certain task decomposition patterns for achieving better performance. For example, the 'tree fan-out' pattern and the 'all to all with nested finish' pattern demonstrate the advantage of decomposing large finish scopes into nested distributed finish scopes assuming a distributed finish implementation is available. The task patterns frequently used in an application should guide the user's choice of a specific TD protocol and implementation.

Moving to the next chapter, we change the focus from improving the efficiency of the resilient async-finish model to improving its productivity. We demonstrate how we exploited the composability and failure awareness capabilities of this model to build resilient abstractions that can reduce the programming effort required for building resilient applications in X10.

Chapter 5

Towards Data Resilience in X10

Multi-resolution resilience — our approach to reconciling resilience, performance, and productivity — depends on the availability of *efficient and composable* low-level resilient constructs that can be used for building productive higher-level resilient constructs. In Chapter 4, we focused on the efficiency of resilient `finish` as a low-level resilient construct for the APGAS model. In this chapter, we focus on its composability. We demonstrate the advantage of providing resilient nested parallelism in the APGAS model for building composable resilient frameworks that can reduce the programming complexity required to create fault-tolerant applications.

Data resilience — the ability to preserve data in spite of failure — is a challenging aspect in the development of resilient applications. In order to remove this burden from the programmer, we designed two resilient stores that can be used by applications for saving critical data or used as a foundation for building higher-level resilient frameworks.

Section 5.1 describes and justifies the adopted design principles for adding data resilience in the APGAS model. Section 5.2 describes a novel extension to the async-finish programming model to support resilient data atomicity that enabled us to provide a transactional resilient store for RX10 applications. Following that, we dedicate Section 5.3 to describing productive resilient application frameworks for X10. In particular, in Section 5.3.1, we describe the implementation details of the two resilient stores; in Section 5.3.2, we describe a resilient iterative framework for bulk-synchronous applications; and in Section 5.3.3, we describe a parallel workers framework suitable for embarrassingly parallel applications. We conclude in Section 5.4 with a performance evaluation using a suite of resilient applications.

This chapter is based on work described in the following publications:

- "A resilient framework for iterative linear algebra applications in X10" [Hamouda et al., 2015], which describes the initial implementation of the iterative application framework.
- "Resilient X10 over MPI user level failure mitigation" [Hamouda et al., 2016], which describes, among other things, enhancements to the performance of iterative applications by exploiting MPI-ULFM capabilities.

- "Failure recovery in resilient X10" [Grove et al., 2019], which is the outcome of a close collaboration with the X10 team on designing and implementing a generic resilient store for X10. While I focused on developing a *native* resilient store that is built entirely using X10 constructs, Olivier Tardieu provided another implementation of the store based on an of-the-shelf resilient store called Hazelcast [Hazelcast, Inc., 2014] for use only by Managed X10. Both the native store and the Hazelcast-based store of X10 *lack support for distributed transactions* [1]. The work described in this chapter related to extending `finish` with atomicity support and the consequent support for distributed transactions was done without significant collaboration with the X10 team. These capabilities have not yet been contributed to the X10 open-source code repository. The work in [Grove et al., 2019] focuses the performance evaluation on Managed X10 and the Hazelcast-based store, except for the LULESH experiment which used the native store. The focus of my thesis is on high-performance computing, therefore I focus my evaluation in this chapter on Native X10 over MPI-ULFM and on the native resilient store. Finally, the iterative framework described in previous publications [Hamouda et al., 2015, 2016] was enhanced by David Grove by integrating it with the `PlaceManager` class (see Section 2.4.5.1) first described in [Grove et al., 2019].

5.1 A Resilient Data Store for the APGAS Model

The APGAS model represents a computation as a control flow of nested parallel tasks and global data partitioned among the places. Failures result in gaps in the control flow and the application data. A resilient APGAS programming model must, therefore, enable restoring a consistent state of both the control flow and the data of the application to enable it to complete successfully after failures. Initial development of RX10 [Cunningham et al., 2014] focused mainly on recovering the control flow and left the burden of protecting application data to the programmer. Although the provided failure-awareness semantics allow implementing a variety of data redundancy techniques, this requires considerable programming effort to ensure the correctness and efficiency of the implementation.

In collaboration with the X10 team in IBM, we addressed this limitation by extending RX10 with a resilient data store abstraction in the form of a distributed concurrent key-value map. We developed two implementations of the store: a place-local store that is suitable for coordinated checkpointing (Section 5.3.1.1) and a transactional store that is suitable for dynamic applications with arbitrary data access patterns (Section 5.3.1.2). Reads and writes to the resilient store can be mixed arbitrarily with normal reads and writes to other memory locations; however, data preservation and atomicity are guaranteed only to the resilient store data.

Our resilient stores provide a high degree of reliability; however, certain failure scenarios, which are rare in practice, can lead to loss of resilient data. We refer to

[1] The Hazelcast-based store of X10 does not expose the transaction functionality available in Hazelcast.

failures that result in losing data from the resilient store as *catastrophic failures*. Our implementations guard each data access operation with validations that guarantee reliable detection and reporting of catastrophic failures.

The following four sections describe the underlying design principles of our APGAS resilient store.

5.1.1 Strong Locality

Aligned with the strong locality feature of the APGAS model, the resilient store is partitioned among the places such that each key-value record explicitly belongs to one place. The store's data at a certain place can be accessed via resilient `get(key)` and `set(key, value)` operations that can nest freely with other X10 constructs. Thanks to the Happens-Before Invariance (HBI) principle underlying the semantics of RX10, the programmer can use the store with the guarantee that, before the failure is raised to the application, orphan tasks updating the store will either complete successfully or never start. In other words, orphan tasks and the application's recovery tasks will not race, which simplifies reasoning about the state of the store's data in the presence of failure.

Each key in the store has a version, a value, and a lock that guards concurrent access to it. The features of the lock are described in Section 5.2.3.2.

5.1.2 Double In-Memory Replication

The storage medium used by the resilient store is a critical factor for determining its survivability and performance. Based on the type of storage medium, resilient stores can be classified into *disk-based* or *diskless* stores.

A disk-based resilient store for an HPC environment typically writes its data in a Parallel File System (PFS), which is available in most large-scale infrastructures. It can survive the failure of the entire computation thanks to the durability of disk storage. However, the high I/O latency of the PFS can slow down data access operations and make the resilient store a performance bottleneck at scale. As the load of disk access increases, the PFS itself can be a source of failures, reducing the reliability of the application [Sato et al., 2014].

A diskless resilient store leverages the low latency of memory access by writing the data in memory while protecting against data loss by employing a data redundancy mechanism, such as replication or data encoding [Cappello, 2009]. The data redundancy mechanism places an upper bound on the survivability of the store. For example, replicating the data at two processes (i.e. double in-memory replication) enables the store to only survive failures that do not kill these two processes at the same time. Increasing the number of replicas improves the store's survivability at a higher performance and memory cost. Because, in practice, most failures impact one or a few nodes simultanously [Lifflander et al., 2013; Sato et al., 2012; Meneses et al., 2012; Moody et al., 2010], a double in-memory resilient store can protect the application from the majority of failures.

We designed the X10 resilient store based on double in-memory replication. As shown in Figure 5.1, each place owns a master replica of its data and a slave replica of its left neighbor's data.

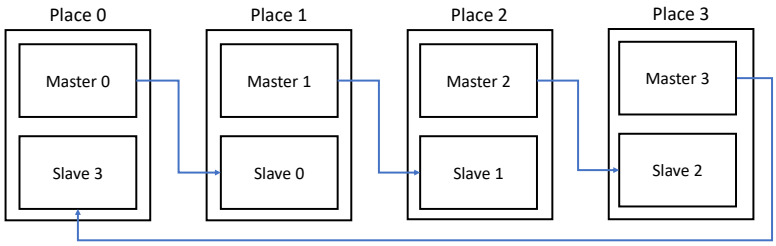

Figure 5.1: Resilient store replication.

5.1.3 Non-Shrinking Recovery

As previously described in our taxonomy in Section 2.2.3.1, resilient runtime systems can support shrinking and/or non-shrinking recovery. For statically partitioned applications, which represent a wide class of HPC applications, shrinking recovery is challenging to use because it requires the programmer not only to adjust the communication topology to accommodate fewer processes, but also to handle possible load imbalance when the workload of the failed process shifts to other processes. Non-shrinking recovery avoids these complexities by resuming the computation on the same number of processes. See Figure 5.2 for a demonstration of shrinking and non-shrinking recovery strategies for a statically partitioned 2D domain.

Our earliest work with RX10, as published in [Hamouda et al., 2015], evaluates the performance of shrinking and non-shrinking recovery for three machine learning benchmarks: linear regression, logistic regression, and PageRank. Linear regression and logistic regression store the input classification examples in a dense matrix partitioned into N horizontal blocks, where N is the number of places. Each block holds 50K classification examples, with 500 features each. PageRank stores the input document graph in a sparse matrix and follows the same partitioning strategy above. Each block holds 2M edges of the graph. The applications achieved data resilience by checkpointing each block both in local memory and in the memory of one neighboring place. Figure 5.3 shows the performance of the three applications after experiencing one place failure using an old version of X10 (v2.5.2).

Non-shrinking recovery results in the fastest performance for the three applications. It maintains the balance between the places and limits the need for communication during recovery to the new place only. Shrinking recovery without repartitioning, in our implementation, assigns the blocks of the failed process to only one live process (similar to the example in Figure 5.2-b). Similar to non-shrinking recovery, only one process needs remote communication for recovery. The rest of the processes recover their blocks from their local memory. Unlike non-shrinking recovery, the load between the places is not balanced, which results in slower performance. Finally,

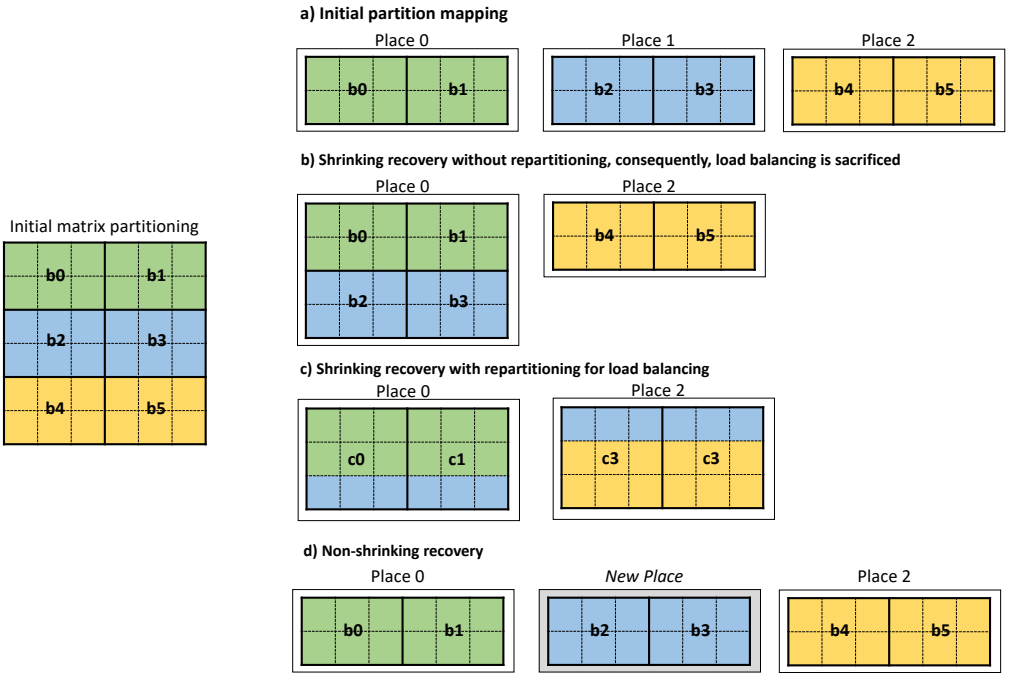

Figure 5.2: Shrinking recovery versus non-shrinking recovery for a 2-dimensional data grid.

shrinking recovery with load balancing results in the slowest performance and the highest performance variability. Changing the block partitioning to achieve load balancing means that the content of a new block may have been distributed among multiple processes before the failure (similar to the example in Figure 5.2-c). During recovery, each place communicates with a group of other places to restore the contents of its blocks, which explains the resulting performance overhead. Many optimizations have been done to RX10 and the iterative framework since publishing this paper. However, the results are still useful as a demonstration of the benefit of non-shrinking recovery for statically balanced bulk-synchronous applications.

Our store supports non-shrinking recovery. Lost replicas due to a place failure are recovered on a spare place using the redundancy available in the store, as shown in Figure 5.4. The `PlaceManager` class provides a logical group of places, named the active places, that maintains a fixed size despite failures as long as spare resources are available. By replacing a failed place with a new place at the same location, it automatically supports place virtualization by allowing the program to use the place order as its identifier, rather than using the physical place id (see Section 2.4.5.1). Our store spans over the group of active places and recovers the data with the awareness that a failed place will be replaced with a new place in the same order.

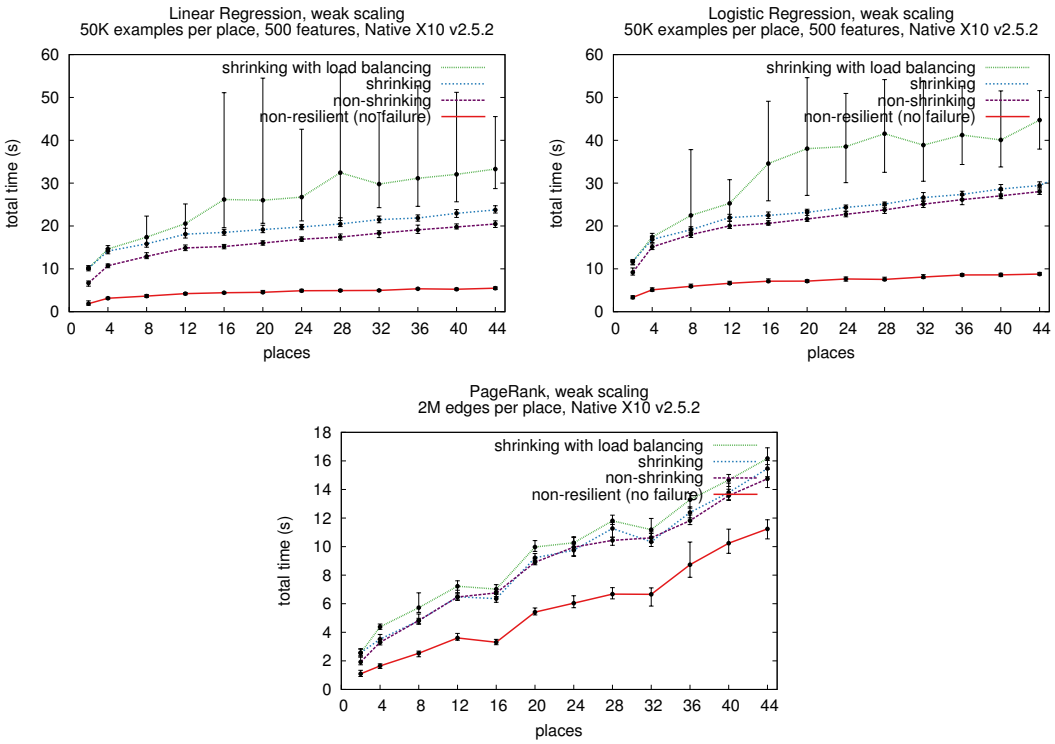

Figure 5.3: Weak scaling performance for three GML benchmarks with non-shrinking and shrinking recovery (see [Hamouda et al., 2015] for more details).

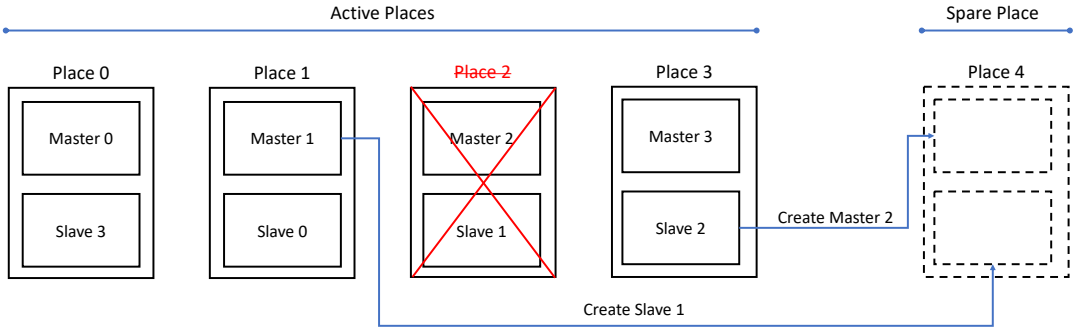

Figure 5.4: Resilient store recovery.

5.1.4 Distributed Transactions

While many applications can use the resilient store for single-place get/set operations, other applications may require atomic multi-place operations. For example, a global load balancing application may require a two-place write operation to steal work from an overloaded place and assign it to an underloaded place atomically. Figure 5.5 shows another example representing a graph clustering problem where parallel tasks perform atomic multi-place write operations to acquire remote vertices to their clusters. Conflicts arise when two parallel tasks attempt to allocate the same vertex (e.g. when task 1 and task 3 attempt to acquire their fifth vertex). The goal is to ensure that the execution of a group of operations on a shared data structure in parallel with other groups is serializable (i.e. equivalent to a serial execution of these groups), and atomic (i.e. either all the operations in a group commit or none.) [Herlihy and Moss, 1993].

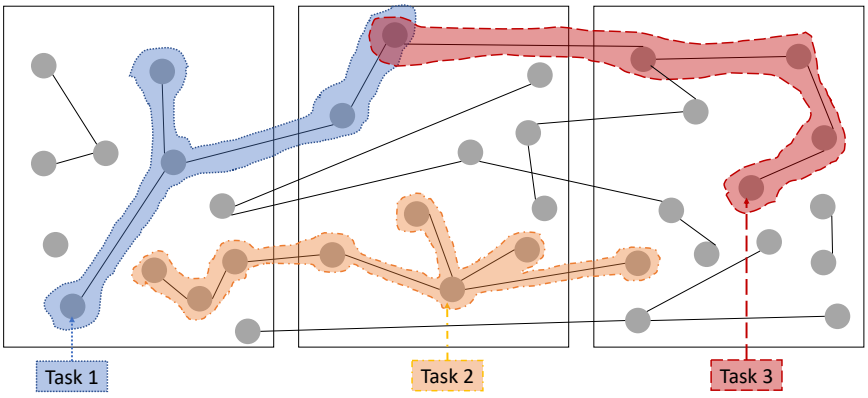

Figure 5.5: Distributed graph clustering example.

A transaction is a productive concurrency control mechanism that aims to hide the serializability challenges from the programmer. The programmer groups a set of read/write operation in a transaction and relies on a Transactional Memory (TM) system to execute it, possibly with other concurrent transactions, while ensuring serializability. A conflict happens when two concurrent transactions access the same data item and at least one is writing. In that case, the TM system aborts at least one of them and rolls back any performed changes. Non-conflicting parallel transactions commit successfully and apply their changes permanently.

A Distributed Transactional Memory (DTM) system [Bernstein and Goodman, 1981] can guarantee serializable execution of transactions spanning multiple nodes (we refer to these nodes as transaction participants). A DTM system relies on two main modules to ensure atomic execution of distributed transactions:

- a concurrency control mechanism executing locally at each participant that can detect conflicts, commit, or abort a piece of a transaction running at a certain participant (see Section 5.2.3.3 for details on our implementation).

- a distributed commit protocol that ensures, via a transaction coordinator, that the effects of a transaction will persist either at all participants or at none, despite the occurrence of failures (see Section 5.2.3.4 for details on our implementation).

Failures impose a major challenge for DTM systems. Without proper failure awareness, a distributed commit protocol may indefinitely delay terminating transactions impacted by a failure as well as other transactions depending on them. To avoid that, DTM systems use fault tolerant commit protocols, which guarantee that every started transaction will eventually terminate despite the failure of a transaction coordinator or any of its participants.

Currently X10 lacks support for distributed transactions, which forces programmers of non-resilient and resilient applications to handle atomic multi-place operations by manually orchestrating data access via locks. However, using locks is prone to programming errors, complicates writing composable codes, and may lead to poor performance if concurrent access is aggressively limited [Harris et al., 2010; Saad and Ravindran, 2011]. These problems are exacerbated when failures are considered.

To enable X10 programmers to easily apply atomic multi-place operations in their programs, we extended X10 with support for distributed transactions over the resilient store (in non-resilient mode, data replication is disabled assuming the absence of failures). A transaction can originate from any place and asynchronously span multiple places using the nested parallelism support of X10. In the next section, we describe the *transactional finish* construct — our approach for combining termination detection and transaction commitment while allowing the full flexibility of nested parallelism. We also detail how we relied on existing mechanisms in the RX10 runtime to provide resilient transactions that can maintain the integrity and availability of the resilient store in the presence of failures.

5.2 From Finish to Transaction

Similarities between the coordination mechanism used by `finish` for termination detection and the coordination mechanism used by distributed commit protocols inspired us to design a combined coordination protocol that achieves both termination detection and distributed transaction commitment. Moreover, it turns out that RX10 already provides the required infrastructure for implementing coordination protocols that are resilient to the loss of a coordinator and/or the loss of any coordinated place.

Many DTM systems tolerate the failure of a coordinator by writing a log of its decisions to disk, assuming that a recovery process will eventually start and terminate the pending transactions using the failed coordinator's log [Mohan et al., 1986]. Other systems avoid delaying transaction termination due to the failure of the coordinator by requiring the participants to elect one of them as a new coordinator [Garcia-Molina, 1982; Keidar and Dolev, 1998]. In both cases, the availability of the coordinator is critical for transaction termination. RX10 has the same requirement for the `finish` objects, and for that reason, it uses a resilient store for maintaining the finish objects.

By providing the finish resilient store, RX10 guarantees the availability of the termination detectors throughout the execution of their tasks, which consequently guarantees that failures will not indefinitely delay the termination of a finish. These guarantees are of course subject to the survivability of the finish resilient store itself. A resilient finish engages in communication with every place used within its scope, which gives it a full and accurate knowledge of its subordinates. Finally, the network layer provides a resilient finish with global failure detection, which enables it to take the necessary recovery actions upon detecting the failure of any of its subordinates. We decided to exploit these powerful capabilities for supporting resilient distributed transactions in RX10.

We propose a *transactional finish* construct that handles, in addition to the normal termination detection, atomic multi-place operations. A transactional finish eventually terminates while guaranteeing data consistency and availability in the absence of catastrophic failures. The following subsections describe the transactional finish in more detail. When we use the term "finish", without preceding it with the word "transactional", we refer to the default finish provided by X10 that is used for synchronization only.

5.2.1 Transactional Finish Construct

Assuming that `Tx` is a class representing a transaction associated with a certain data store, we propose the construct **`finish`** `(tx:Tx) S;`, which is used as follows:

```
try { finish (tx:Tx) S; } catch (e:MultipleExceptions) { E; }
```

By parameterizing a finish with a `Tx` instance, this finish works as a transactional finish that not only waits for the termination of all asynchronous tasks created by `S`, but also ensures that all data access operations, in the form of `tx.get(key)` and `tx.set(key, value)`, invoked by `S` are executed atomically.

The runtime system implicitly creates an instance of type `Tx` and attaches it to the **`finish`**. Each transaction has a globally unique identifier accessed by `tx.id`.

5.2.1.1 Nesting Semantics

A transactional finish can include within its scope nested invocations of **`finish`**, **`async`** and **`at`**. Although syntactically, we also allow a transactional finish to include nested transactional finishes, our current implementation lacks proper support for nested transactions [Moss and Hosking, 2006]. Therefore, the runtime system currently executes nested transactions as independent transactions. We aim to address this limitation in the future by exploring the diverse mechanisms proposed in the literature for handling nested transactions.

The code in Listing 5.1 demonstrates the nesting feature of a transactional finish. The transactional finish starting at Line 1 spawns three asynchronous tasks at places p, s, and t. Place p sets $D = 100$, and place q sets $C = A + B$, where A is hosted at place r, and B is hosted at place s. The finish at Line 5 is a synchronization-only finish used to ensure that reading A and B completes before C is updated. Place t

does not perform any updates on the data store. X10's **atomic** construct provides atomicity protection within a place. It is used in Line 8 and Line 12 to guard from race conditions due to updating the shared variable $gr()$ by the two asynchronous tasks created by place r and place s (see Section 2.4.3 for a description of `GlobalRef`).

Listing 5.1: Transactional finish code example.

```
1 finish (tx:Tx) {
2     at (p) async tx.set("D", 100);
3     at (q) async {
4         val gr = GlobalRef[Box](new Box());
5         finish {
6             at (r) async {
7                 val a = tx.get("A");
8                 at (q) async  atomic {  gr().add(a); }
9             }
10            at (s) async {
11                val b = tx.get("B");
12                at (q) async atomic {  gr().add(b); }
13            }
14        }
15        tx.set("C", gr().getValue()); // the sum of a and b
16    }
17    at (t) async { print("Hello"); }
18 }
```

The transactional finish at Line 1 is the transaction coordinator. After all nested tasks complete, it implicitly starts a commit protocol that includes places p, q, r, and s only. There is no need to include place t because it does not perform any read or write operation on the data store. Normally, an outer finish lacks the knowledge about the places visited within the scope of an inner finish. However, as a transaction coordinator, a transactional finish must know all the places that accessed the data store either directly or through a nested finish. Section 5.2.2.2 describes how we enable a transactional finish to gain this knowledge.

5.2.1.2 Error Reporting Semantics

Like a normal **finish**, a transactional **finish** also records the exceptions thrown within its scope and reports them through an instance of type `MultipleExceptions`. A transactional **finish** that terminates successfully without an exception indicates a successfully committed transaction. In contrast, a transactional **finish** that terminates with an exception indicates an aborted transaction. The root cause of aborting the transaction can be detected by inspecting the list of the reported exceptions. The list can include: 1) instances of type `DeadPlaceException` (`DPE`) reporting the failure of a used place (can be a participant or a non-participant in the transaction), 2) instances of type `ConflictException` (`CE`) reporting conflicts detected at participants, or 3) any other application-related exceptions.

While the user can catch exceptions thrown by an inner **finish**, doing so does

not prevent the parent transactional finish from aborting the transaction if any of the participants died or experienced a conflict. Therefore, the use of **finish** within a transaction should be for synchronization purposes only and not for handling `CEs` or `DPEs`.

In resilient mode, committing a transaction attempts to apply its updates on the master and slave replicas of each participant. Depending on the specification of the commit protocol, a transactional finish can mask a failure that impacts one of the replicas assuming that the surviving replica will recover the lost replica using its own state. Section 5.2.3.4 describes how our implementation handles the failure of one of the replicas during the commit protocol.

5.2.1.3 Compiler-Free Implementation

To avoid the complexities involved in modifying the X10 compiler, we currently provide a compiler-free implementation of the transactional **finish** construct. It requires manual construction of a `Tx` object and manual binding of this object to a **finish** as shown in the code below. The method `store.makeTransaction` creates a `Tx` object, and the method `Runtime.registerTransaction` registers a transaction for a finish. The `store` variable is an instance of the `TxStore` class that we will describe in Section 5.3.1.2.

```
1 val tx = store.makeTransaction();
2 finish {
3    //must be the first action in the finish block
4    Runtime.registerTransaction(tx);
5    S;
6 }
```

5.2.2 Finish Atomicity Awareness

A transactional **finish** is a special type of finish that acts as a distributed transaction. It runs through two phases: execution and commitment. The execution phase is when tasks are being evaluated and spawned within the **finish** scope. When all tasks terminate, the commitment phase starts by invoking a commit protocol to terminate the transaction by either committing it or aborting it. Table 5.1 describes three differences between a **finish** and a transactional **finish** — all the differences are related to tracking the participants of a transaction. In the following, we describe how we amended the finish protocol to enable accurate tracking of transaction participants.

5.2.2.1 The Join Signal

While finish considers all places equal, a transactional finish distinguishes transaction participants from non-participants. This distinction enables it to speed up the commitment phase by ignoring non-participants. According to Mohan et al. [1986], one of the desirable characteristics in a commit protocol is "exploitation of completely or partially read-only transactions". Thus, a transactional finish aims to distinguish

Table 5.1: Comparison between finish and transactional finish.

	Finish	Transactional Finish
1	all visited places are equal	performance optimizations can be achieved by 1) distinguishing participants from non-participants and 2) distinguishing read-only participants
2	finish can forget a place once all its tasks have terminated	a transactional finish must not forget a transaction participant
3	an outer finish is unaware of places visited by an inner finish (an adopted pessimistic finish is an exception)	a transactional finish must identify all participants, even those visited by an inner finish

read-only participants from write participants to exploit more opportunities for performance optimizations.

We used the 'join' signal as a means of identifying the transaction participants and their types. Each task is modified to hold two properties: whether it accessed the resilient store, and if so, whether it accessed the store for reading only. These properties are implicitly set by `tx.get(key)` and `tx.set(key, value)` when called by a task. When a task terminates, it includes these properties as part of its 'join' signal to the coordinating finish, which uses this information to identify the participants and their types.

Depending on the termination detection protocol, it may be possible for a transactional finish to forget the identity of a non-participant to save memory; however, it must keep the identities of the participants available until after the commitment phase completes.

5.2.2.2 The Merge Signal

An inner finish normally hides details about the places it is tracking from its parent. If this inner finish is part of a transaction, it may be tracking a subset of the transaction participants, and in that case, it must share this knowledge with the transaction coordinator (which is the nearest outer transactional finish). We added the 'merge' signal to achieve this goal.

Each finish is modified to hold the following properties: a property to indicate whether the finish is part of a transaction, a property to indicate whether the finish is the root of a transaction, and a property that holds a `Tx` instance. The `Tx` instance records the participants (if any) and their types. A finish that is not part of a transaction or that did not detect any transaction participant terminates normally. Otherwise, if this finish is not the root of the transaction, it shares the list of participants with its direct parent by sending a 'merge' signal to it. The chain of finish objects recursively delivers the participants set to the nearest transactional finish construct. In resilient mode, details about the transaction participants are stored as part of the resilient finish object which is expected to survive failures.

5.2.2.3 Extended Finish Protocols

We successfully extended the non-resilient finish implementation and the optimistic finish implementations (place-zero and distributed) with atomicity awareness as described above. We did not use the pessimistic finish protocol, not only because it is slower than optimistic finish, but also because its adoption mechanism is more complex. In the pessimistic protocol, when a finish is lost, its corresponding resilient finish is disabled, and the tasks associated with it are merged with the tasks of its parent. If the lost finish is a transaction, the parent finish must carry the responsibility of terminating the transaction on behalf of the lost finish. The optimistic protocol, on the other hand, does not disable the resilient finish object corresponding to a lost finish. It keeps it active as a ghost finish that continues to carry the responsibility of tracking its tasks (see Section 4.9.1), and if it is a transaction, it handles the transaction commitment as usual. We found this approach simpler to implement and easier for integrating the transaction-related changes.

5.2.3 Implementation Details

5.2.3.1 Transaction Identifier and Log

Each transaction is assigned a globally unique eight-byte transaction identifier. The first four bytes store the physical id of the transaction coordinator place. The second four bytes store a place-local sequence number.

The transaction log maintains the readset and writeset of the transaction. Each participant holds a partition of the log describing the set of read and written keys in its place. The transaction log is used by the local concurrency control (CC) mechanism for tracking used keys and detecting conflicts (see Section 5.2.3.3). In section 5.2.3.4, we explain how the transaction log is central to tolerating the failure of a participant during the execution of the commit protocol.

5.2.3.2 Lock Specification

The internal implementation of our concurrency control mechanism is lock-based. We assign an upgradable read-write lock to each key. Locking and unlocking are performed according to the CC mechanism used by the store, which we will describe in Section 5.2.3.3.

Lock-based TM systems often employ a deadlock detection or avoidance scheme to guarantee system progress [Moss, 1981]. Deadlock avoidance schemes are more attractive in distributed systems as they avoid the extra communication required in deadlock detection schemes to discover circular dependencies between transactions. Wait-Die and Wound-Wait are two types of locks that can be used for deadlock avoidance [Rosenkrantz et al., 1978]. The first word refers to the action the older transaction makes, and the second word refers to the action the younger transaction makes. In the wait-die lock, an older transaction waits if a younger transaction is acquiring the lock, and a younger transaction dies if an older transaction is acquiring the lock. In the wound-wait lock, an older transaction forces a younger transaction to

release the lock, and a younger transaction waits if an older transaction is acquiring the lock.

We prefer wait-die over wound-wait as we found it easier to design our system with the knowledge that a transaction that acquired a lock will continue to hold it until it voluntarily commits or aborts. Otherwise, with wound-wait, we need to check if the lock is still held before accessing every key that we previously acquired. Thus our lock can be described more precisely as an *upgradable read-write wait-die lock*.

A reader or a writer waits only if the lock is being acquired by a younger writer or reader, respectively. A waiting transaction aborts when an older transaction requests the lock. A waiting writer has a higher priority than a waiting reader.

The priority of a transaction is decided based on its identifier. When comparing transactions to find out which one is older, we first compare their place-local sequence number. If the sequence numbers are identical, we compare the place numbers. The transaction with the smaller sequence number or place number is considered the older transaction.

5.2.3.3 Concurrency Control Mechanism

The essence of a CC mechanism can be captured by describing how reads and writes are performed. Bocchino et al. [2008] provide a comprehensive list of design aspects for TM systems, three of which relate to handling reads and writes: read synchronization (read versioning or read locking), write acquire time (early or late), and write recovery (undo-logging or write-buffering). In the following paragraphs, we describe these choices based on the description in [Bocchino et al., 2008] and how they are implemented in our store.

Read Versioning vs. Read Locking

In Read Versioning (RV), a version is assigned to each key. Each reading transaction records the version of the key the first time it reads the key. It compares that version with the current version of the key at commit preparation time. A different version indicates a conflict: another transaction has updated the key since it was read, and the current transaction must abort. Read Locking (RL) prevents other transactions from writing by acquiring the key lock at read time. A writer may not acquire the lock until all readers have released it.

Early Acquire vs. Late Acquire

In Early Acquire (EA), a writing transaction acquires an exclusive lock to the key the first time it writes the key. Late Acquire (LA) delays acquiring the lock until commit preparation time. In both cases, if RL is used and the key was previously locked for read, the lock is upgraded from read to write. If RV was used, the key version is compared directly after acquiring the write lock. If the version is different, the transaction aborts.

Write-Buffering vs. Undo-Logging

These are two methods for recovering failed writes. In Write-Buffering (WB), a transaction performs its writes on a shadow copy of the value. Subsequent reads from the same transaction return the buffered value. Changes made by one transaction are not visible to other transactions until the change is committed. Write buffering can be used with EA or LA. In Undo-Logging (UL), writes are done in-place. The transaction records the old value to undo its changes in case of abort. Undo-Logging can only be used with EA.

According to the above choices, six different mechanisms are formed: RL_EA_UL, RL_EA_WB, RL_LA_WB, RV_EA_UL, RV_EA_WB, RV_LA_WB. We implemented the six mechanisms to give the flexibility of selecting a suitable mechanism based on the workload properties.

5.2.3.4 Two Phase Commit

A transactional **`finish`** starts the commit protocol only after all tasks have terminated and the full list of participants has been captured. In this section, we describe two variants of the Two-Phase Commit (2PC) [Mohan et al., 1986] protocol that we used for coordinating transaction termination in non-resilient and resilient modes.

In order for a commit protocol to guarantee reaching a consistent decision at all participants, the following requirements must be satisfied. We quote these requirements from [Keidar and Dolev, 1998], who obtained this list from Chapter 7 of [Bernstein et al., 1987].

> **AC1**: Uniform Agreement: All the sites that reach a decision reach the same one.
>
> **AC2**: A site cannot reverse its decision after it has reached one.
>
> **AC3**: Validity: The commit decision can be reached only if all sites voted Yes.
>
> **AC4**: Non-triviality: If there are no failures and all sites voted Yes, then the decision will be to commit.
>
> **AC5**: Termination: At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long, then all sites will eventually reach a decision.

Non-Resilient 2PC Protocol

Assuming the absence of failures, the 2PC protocol satisfies the above commitment requirements using two phases. In the first phase, the coordinator sends a `PREPARE` message to all participants in parallel asking them to vote on whether to commit or abort the transaction. Each participant validates its transaction log according to the used CC mechanism to check for conflicts. If the participant detects a conflict,

it aborts the transaction locally and votes to abort, otherwise, it votes to commit. If all participants voted to commit, the coordinator sends a `COMMIT` message to all participants. Otherwise, the coordinator sends an `ABORT` message to participants that voted to commit (the other participants have already aborted before sending the abort vote). The transaction is considered terminated only after the coordinator receives acknowledgements for all the `COMMIT/ABORT` messages. Figure 5.6 and Figure 5.7 demonstrate the protocol for a committed transaction and an aborted transaction, respectively.

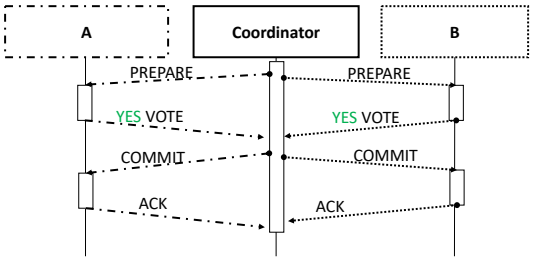

Figure 5.6: Two-phase commit: a committing transaction.

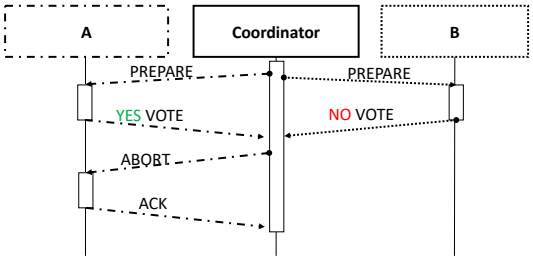

Figure 5.7: Two-phase commit: an aborting transaction.

We exploit the following optimizations in the non-resilient protocol:

- Checkpointing the decisions taken by the coordinator or the participants is not needed since all the parties are assumed to be alive throughout the commit protocol.
- As described in [Bocchino et al., 2008], the prepare phase may be omitted in CC mechanisms that eagerly acquire locks of accessed keys, namely the RL_EA_*. These mechanisms detect the conflicts during the execution phase. Completing the execution phase successfully indicates to the coordinator that all participants are willing to commit, and consequently it can start the second phase directly. The mechanisms that use RV attempt to acquire the read locks in the prepare phase. Similarly, mechanisms that use LA attempt to acquire the write locks in the prepare phase. Therefore, the prepare phase cannot be omitted in these mechanisms.

- There is no special handling for read-only transactions.

Fault Tolerance Features

Failures have critical impacts on the availability of a DTM system. A failed coordinator may leave behind active participants that acquired data locks and are waiting for the transaction to progress to release these locks. Not only would this transaction hang, but would other transactions that depend on the acquired locks. A failure of a participant results in data loss if data replication is not employed. Meanwhile, ensuring replication consistency in the presence of failures is challenging to achieve.

We addressed these complexities in a resilient variant of the 2PC protocol that aims to guarantee transaction atomicity on the resilient APGAS data store. Our design relies on the finish resilient data store to guarantee the survivability of the transactional `finish` objects serving as transaction coordinators. Our design also exploits the replication mechanism of the APGAS data store to protect data availability.

A common mechanism to allow a 2PC protocol to tolerate the loss of a coordinator or its subordinates is by checkpointing critical transaction decisions in disk. For example, in [Mohan et al., 1986], during the prepare phase, a participant persists its prepare decision on disk to ensure that a recovering process will not reverse this decision (to satisfy requirement **AC2**). A participant that is prepared to commit also persists a prepare-log with the data changes required should the transaction commit. Similarly, the coordinator persists the decision to commit or abort the transaction on disk to force the recovering coordinator to take the same decision. We follow the same strategy; however, rather than persisting on disk, we persist critical data in two in-memory replicas.

We start by describing the resilient 2PC protocol in normal operation when failures are absent, then we describe how the protocol handles failures.

Resilient 2PC Under Normal Operation

The protocol can be described with respect to four actors: the coordinator, the backup coordinator, the participant's master, and the participant's slave. While the transaction is executing, the only actors that are used are the coordinator and the masters of the participants. In the commitment phase, the other actors are also used subject to certain conditions.

The coordinator uses its local knowledge of the active places in the system to build a presumed master-slave mapping. This mapping may be inaccurate if one of the masters asynchronously recovered its slave replica. The coordinator (or the backup coordinator) detects this inaccuracy through the masters in the prepare phase, and consequently aborts the transaction. Retrying this transaction will eventually succeed when the coordinator's place detects the new slave replica (see the Handshaking description in Section 5.3.1.2).

Each participant can accurately identify the coordinator's place from the transaction id. However, they cannot accurately identify the place of the backup coordinator. Knowing the places of the coordinator and its backup enables a participant and its

slave to detect a catastrophic failure when both places fail. We require the coordinator to share the place of its backup with the participants during the prepare phase.

Assuming a full-write transaction, in which all participants are writers, the committment protocol works as follows. In the first phase, the coordinator sends a `PREPARE` message to each master carrying the identity of the slave known to the coordinator and the identity of the backup coordinator. In return, the coordinator expects to receive two acknowledgements — one from the master and one from the slave. The master place can take one of the following two paths when handling a `PREPARE` message:

- If the master detects that the given slave's identity is incorrect (because the old slave died and a new slave was created) or if the master detects a conflict, it aborts the transaction and sends a message with two acknowledgements — one for itself to send the abort vote and one emulating an acknowledgement from the dead slave. An aborted master and its slave are ignored in the second phase.

- If the master detects that the slave's identity is correct and that no conflicts exist, it prepares to commit by sending two messages — a message to its slave carrying a **prepare-log** and the identity of the backup coordinator and a message to the coordinator to vote for commit. The slave stores the backup's identity and the prepare-log, which includes, for each modified key, the expected version before write and the new value. After that, the slave sends an acknowledgement to the coordinator. The provided version number for each key is used by the slave at commit time only to guarantee committing the transactions in the same order applied by the master.

The second phase starts after the coordinator receives two acknowledgements for each participant.

If all masters voted to commit, the coordinator sends a **commit-log** to its backup that carries the commit decision and the master-slave mapping. After receiving an acknowledgement from its backup, the coordinator sends direct `COMMIT` messages to each master and slave and waits for acknowledgements from them. Because the coordinator sends the `COMMIT` message to a master and its slave in parallel, it is possible that the master commits the transaction, releases the transaction locks, and starts a second transaction on the same keys, before the slave commits the first transaction. If the master prepares the second transaction to commit before the slave commits the first transaction, the slave will hold two transaction logs that may be altering the same keys. If the second transaction aborts, the risk of inconsistency disappears. However, if the second transaction commits, the slave may receive the `COMMIT` message of the second transaction before the first. Using the versions of the keys, the slave detects this problem and handles it by committing the first transaction before the second. Thanks to the lock-based CC mechanism applied by the master, which ensures that the version of a key is incremented only after its transaction commits, requesting a slave to commit a key with version $v + 1$, occurs only if the transaction that sets the version to v has committed. This guarantee makes it safe

for the slave to commit the first transaction even before receiving its `COMMIT` message. The master and the slave acknowledge the coordinator after handling the `COMMIT` message.

If any of the masters voted to abort, the coordinator decides to abort the transaction without notifying the backup coordinator (the backup coordinator restarts the prepare phase to recover any non-committed transaction). The coordinator sends direct `ABORT` messages to each master and slave that voted to commit and waits for acknowledgements from them. The transaction terminates after receiving the required acknowledgements. At this point, the coordinator sends a message to delete the backup coordinator. Figure 5.8 and Figure 5.9 demonstrate the protocol for a committed transaction and an aborted transaction, respectively.

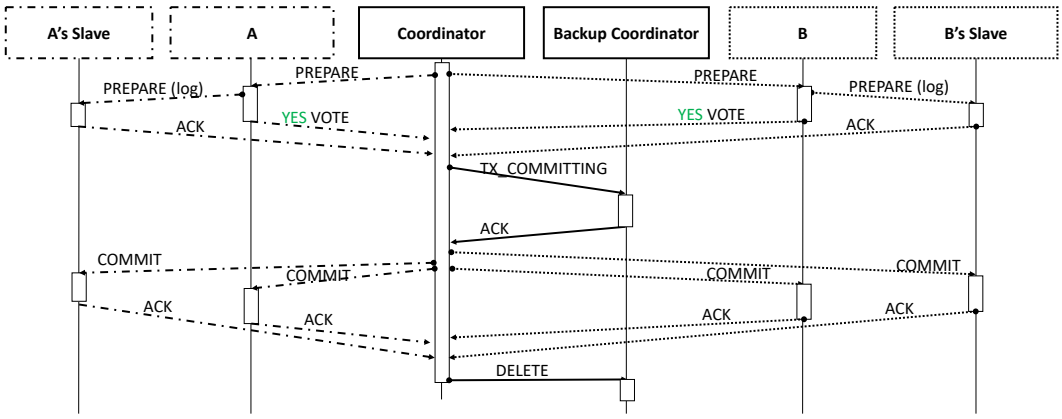

Figure 5.8: Resilient two-phase commit: a committing write transaction.

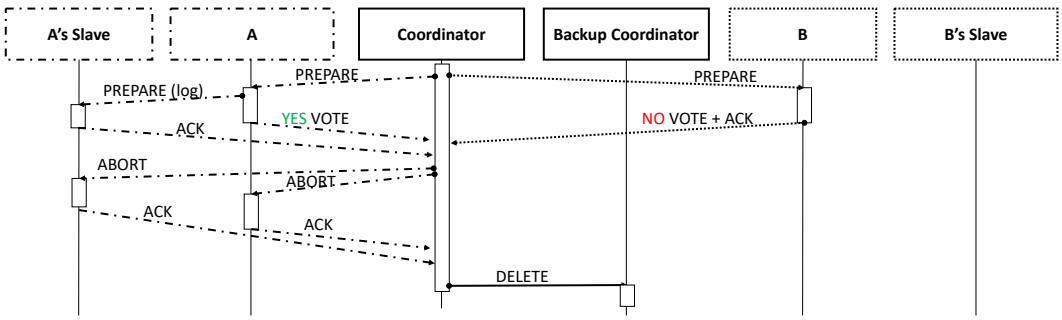

Figure 5.9: Resilient two-phase commit: an aborting write transaction.

We exploit the following optimizations in the resilient protocol:

- In the prepare phase, a read-only participant does not need to generate a prepare-log because none of its actions require updating the store. Therefore, the slave of a read-only participant is completely ignored by the coordinator.
- While in the non-resilient protocol, the prepare phase can be omitted when RL_

EA_* is used for both read-only and write transactions, in the resilient protocol, PREPARE can only be omitted in read-only transactions. That is because the purpose of PREPARE in the resilient protocol is not limited to detecting conflicts, but it also replicates the prepare-log of the write operations.

Note that the implementation of the resilient finish store determines how the backup coordinator will be created. In the place-zero resilient store, the coordinator and its backup are basically the same object saved at place-zero. In the distributed resilient store, the coordinator is the master finish object located at the finish home place, and the backup coordinator is the backup finish object located at the next place.

Resilient 2PC Under Coordinator Failure

When the place-zero resilient finish store is used, we assume that the coordinators will never fail. That is because all coordinators are hosted at place-zero, which is assumed to survive failures. Therefore, we focus the following discussion on the distributed finish store that creates for each finish object, a backup finish at the next place. In this store, a coordinator failure occurs when the place from which the transaction has originated fails before the transaction terminates. As long as the failed place is not place-zero and the backup place is alive, coordinator recovery is possible.

By default, transaction commitment is handled by the master finish object representing the transaction coordinator. A coordinator can be lost during the execution phase of a transaction before 2PC starts. When the optimistic finish protocol is used, a failed coordinator that is in the execution phase is recreated at another place using the backup coordinator's state (i.e. backup finish object). The new coordinator completes the execution phase and then handles transaction commitment normally.

The loss of a coordinator during the commitment phase makes the backup coordinator in charge of transaction commitment. If the backup coordinator received a commit-log from the master finish, it starts the second phase of the 2PC protocol by sending a COMMIT message to all participants and their slaves. Otherwise, the backup coordinator starts from the first phase by sending the PREPARE messages.

The participants are prepared to handle duplicated messages due to coordinator recovery under the following rules:

- A committed master must not receive a duplicated PREPARE message. That is guaranteed using the commit-log that is given to the backup coordinator before any place commits.
- An aborted master may receive a duplicated PREPARE message. Handling the first message concludes by deleting the transaction log. The absence of a transaction log directs the place to vote to abort.
- A committed (An aborted) master or slave may receive a duplicated COMMIT (ABORT) message. Handling the first message concludes by deleting the transaction log. The absence of a transaction log directs the place to ignore the duplicated message.

Resilient 2PC Under Backup Coordinator Failure

Assuming the coordinator is active, the backup coordinator may also fail during the execution phase or the commitment phase. A failure during the execution phase is handled by recreating the backup coordinator at another place. A failure during the commitment phase can only be detected by the coordinator while sending the commit-log. We ignore this failure and allow the coordinator to commit the transaction assuming that a coordinator's failure is unlikely to occur within a short time interval from the failure of its backup.

Resilient 2PC Under Participant Failure

When a transaction commit is initiated, the transaction coordinator builds a master-slave mapping for the participants and checks their status. The failure of both a master and its slave is an indication of a catastrophic failure. Otherwise, the commit protocol proceeds as usual. During the two phases of the commit protocol, the coordinator blocks to wait for acknowledgements from masters and slaves. Using failure notifications from the runtime system, it can adjust the number of pending acknowledgements to avoid waiting for them unnecessarily.

During the prepare phase, the following rules are used for handling master and slave failures:

- On detecting the failure of a master, the coordinator drops the pending acknowledgement of the master and its slave. The reason why we drop the slave's acknowledgement although it is still alive is because the slave joins the prepare phase through its master (by receiving the prepare-log). If the master dies before sending the prepare-log to the slave, the slave will never acknowledge the coordinator. The coordinator drops the slave's acknowledgement if it was still pending to avoid waiting for a message that may never be sent.
- On detecting the failure of a slave, the coordinator drops the slave's pending acknowledgement.
- The failure of a master or a slave during the prepare phase causes the transaction to abort.

In the second phase, the coordinator communicates directly with the masters and the slaves (of the writing masters only) to send the `COMMIT` or `ABORT` messages. On detecting the failure of a master or a slave, it drops the associated pending acknowledgement and proceeds normally, as long as at least one replica for each participant is alive.

We rely on the replication recovery mechanism of the store to recover the lost replica using the consistent content of the other replica. A replica is considered to be in an inconsistent state if it holds unterminated transactions. The store must wait until all unterminated transactions on the replica terminate by committing or aborting before starting the recovery process. The coordinator does not prevent a transaction from terminating at one replica if the other replica has failed.

If the transaction is committing, the coordinator masks failures during the second phase from the programmer, and the transactional finish construct completes successfully. If the transaction is aborting, the programmer receives a `MultipleExceptions` error that includes `DPEs` for the failed places.

5.2.3.5 Transaction Termination Guarantee

We claim that, in the absence of a catastrophic failure, any started transaction will eventually terminate. The rationale behind this claim is based on the following guarantees provided by RX10 for detecting failures and protecting the finish objects and the described resilient 2PC protocol:

1. Failure detection: the failure of any place is eventually detected by the RX10 network layer.

2. Coordinator availability: with a place-zero finish resilient store, a coordinator is assumed to never die because the failure of place-zero is catastrophic. The distributed resilient store provides, for each finish, a backup finish at another place that may recreate the master (if the transaction is in the execution phase) or takeover terminating the transaction (if the transaction is in the commitment phase). In both cases, there is always a coordinator in charge of terminating any started transaction.

3. Master and slave migratability: recovering a lost replica is performed by migrating the data of the surviving replica to a new place. The surviving replica cannot migrate its content if it is in the middle of handling transactions that can alter its state. It needs to reach a state in which all pending transactions that have the potential to modify its data have completed. Because every started transaction is guranteed to terminate thanks to the coordinator's availability gurantee (point 2), a replica aiming to migrate will reach a migratable state within a bounded waiting time.

Unfortunately, due to time limitation, we could not develop a formal model of the transactional finish to prove its correctness within the time-frame of the thesis. Our design mainly applies mechanisms that are widely studied in the literature, i.e. 2PC and strong-consistency of replicated data, without innovating new features to them. Assuming the correctness of these mechanisms, the correctness of the optimistic finish protocol (see Section 4.9.4), the correctness of the finish replication protocol (see Section 4.10), and the soundness of the measures we took to guarantee transaction termination and master/backup consistency, we believe that we have covered the vast majority, if not all, of the flaws that may cause incorrect behavior. Proving that formally is left as a possible future extension to this work.

Based on the above features, we designed a transactional store application framework for X10 and RX10 applications. The store implements an asynchronous recovery mechanism that is based on the above transaction termination guarantees. We will describe the transactional store framework and the used replica migration protocol

in Section 5.3.1.2. For applications that do not require asynchronous recovery and atomic multi-place operations, we designed a simpler resilient store that avoids all transaction handling complexities. We describe this simple store in Section 5.3.1.1.

5.3 Resilient Application Frameworks

In this section, we demonstrate X10's multi-resolution resilience by building resilient frameworks at different levels of abstraction. Using finish as a low-level construct, we built two resilient stores. We then built two application frameworks, based on the resilient stores, that hide most of the fault tolerance details from the user.

All the frameworks support non-shrinking recovery, by replacing a failed place with a new place. We utilize X10's `PlaceManager` utility, provided by the X10 team, to manage the group of active places available to the application. As a reminder, we list the `PlaceManager` APIs in Table 5.2. Detailed description of these APIs can be found in Section 2.4.5.1. X10 uses the keyword **`this`** for defining the class constructor.

Table 5.2: The PlaceManager APIs.

Class	Function	Returns
PlaceManager	this(numSpares:Long, allowShrinking:Boolean)	PlaceManager
	activePlaces()	PlaceGroup
	rebuildActivePlaces()	ChangeDescription
ChangeDescription	addedPlaces()	PlaceGroup
	removedPlaces()	PlaceGroup

5.3.1 Application Resilient Stores

5.3.1.1 Place Local Store

The `PlaceLocalStore` class provides a resilient store that a place can use to save its local data resiliently. It saves the data of each place in a master replica located at the same place, and a slave replica located at the next place. The store's APIs are summarized in Table 5.3 and described here:

- `make(activePlaces)`: creates a store for the given group of places.
- `set(key, value)`: atomically updates the master and the slave replicas of the calling place with the provided key-value pair.
- `get(key)`: returns the value associated with the given key.
- `recover(changes)`: recovers the store given a description of the changes impacting the places.

The store does not support distributed transactions because it is intended for use for single-place operations only. Its recovery is user-driven and global; therefore, it is most convenient for use by bulk-synchronous applications. Fortunately, the resilient **finish** construct provides all the capabilities required to implement the above functions, as we will describe below.

Table 5.3: The PlaceLocalStore APIs.

Class	Function	Returns
PlaceLocalStore[K,V]	make(activePlaces:PlaceGroup)	PlaceLocalStore[K,V]
	get(key:K)	V
	set(key:K, value:V)	void
	recover(changes:ChangeDescription)	void

Resilient Atomic Updates Using Finish

We use synchronous replication to guarantee that updates to the store are applied in the correct order on both replicas. Using **finish**, we ensure that each update is applied at the slave replica before applying it at the master replica. A failing slave causes **finish** to throw an exception that skips over the update operation of the master, as shown in the code below:

```
1 def set(key:K, value:V) {
2     finish at (slave) async slave.set(key, value);
3     master.set(key, value);
4 }
```

The `get` operation uses the master replica only; therefore, it does not involve any communication. Parallel invocations of `get` and `set` requests are serialized at the master replica using an exclusive lock.

Replication Recovery Using Finish

The `PlaceLocalStore` applies a simple recovery mechanism that is convenient for bulk-synchronous applications. When some places fail, the program is expected to enter a recovery phase in which updates to the store are avoided. One place, usually the first place, uses the `PlaceManager` to rebuild the active places and obtain a description of place changes. The new group of active places will include previously uninitialized spare places. To initialize the spare places, the program invokes `recover(changes)`, which initializes each spare place with a master replica (obtained from its right neighbor) and a slave replica (obtained from its left neighbor), as outlined in Listing 5.2.

The recovery procedure starts by checking that at least one replica of each place is still available (Line 2). Violation of this condition results in throwing a fatal error to the application. The finish at Line 4 waits for all recovery tasks to complete and reports any detected failures while initializing the spare places. After successfully

recovering the store, the application proceeds with the guarantee that saved data before the failure are available.

Listing 5.2: PlaceLocalStore recovery.

```
1 def recover(changes:ChangeDescription) {
2     checkIfBothMasterAndSlaveDied(changes);
3     var i:Long = 0;
4     finish for (deadPlace in changes.removedPlaces) {
5         val spare = changes.addedPlaces.get(i++);
6         val left = changes.oldActivePlaces.prev(deadPlace);
7         val right = changes.oldActivePlaces.next(deadPlace);
8         //create the spare's master replica
9         at (right) async slave.copyToMaster(spare);
10        //create the spare's slave replica
11        at (left) async master.copyToSlave(spare);
12    }
13 }
```

The `PlaceLocalStore` simplifies the implementation of coordinated and uncoordinated checkpointing protocols by hiding the replication and recovery complexities from the programmer. However, global recovery is inefficient and difficult to implement for dynamic applications where synchronization between the places is limited or unnecessary. In Section 5.3.1.2, we describe a more sophisticated store that supports asynchronous recovery of the places and also supports distributed transactions.

5.3.1.2 Transactional Store

The `TxStore` class provides a resilient store that supports distributed transactions and ensures data consistency and availability in the presence of non-catastrophic failures. It supports both global synchronous recovery (in the same way applied in `PlaceLocalStore`) and local asynchronous recovery. Table 5.4 lists the store's APIs.

Table 5.4: The TxStore APIs.

Class	Function	Returns
TxStore[K,V]	make(pg:PlaceGroup, asyncRec:Boolean, cc:String, work:(TxStore[K,V])=>void)	TxStore[K,V]
	activePlaces()	PlaceGroup
	recover(changes:ChangeDescription)	void
	makeTransaction()	Tx[K,V]
	executeTransaction(closure:(Tx[K,V])=>void)	void
Tx[K,V]	get(key:Key)	V
	set(key:K, value:V)	void
	asyncAt(virtualId:Long, closure:()=>void)	void

The program creates an instance of `TxStore` by calling `TxStore.make` and providing the initialization parameters. The first parameter `pg` is the initial group of active places, the second parameter `asyncRec` states whether asynchronous recovery should be used, the third parameter `cc` states the chosen concurrency control mechanism (see Section 5.2.3.3), and the fourth parameter `work` defines the work to be done by a new joining place (used in asynchronous recovery only). We will show how the `work` closure is invoked during the recovery in Listing 5.3-Line 29.

Each active place receives a copy of the activePlaces group at start-up time. The method `activePlaces()` evaluates locally and returns the activePlaces known to the current place at the time of invocation. When asynchronous recovery is used, updates to the active places occur asynchronously and are not reflected at all places at the same time. Thus, calling `activePlaces()` at different places may return different values depending on what updates each place discovered.

When synchronous recovery is used, place failure events are reported to the application through exceptions. The application may react to these failures by calling `recover(changes)` to initiate a global synchronous recovery in the same way described in Section 5.3.1.1.

Transaction Processing

All read and write operations on the store are invoked by transactions. The function `makeTransaction` creates a new transaction of type `Tx`. The `executeTransaction` function executes a closure representing the body of a transaction within a transactional finish scope. Although it is acceptable for the application to execute the transaction directly, `executeTransaction` provides the advantage of handling transaction exceptions transparently. It captures errors raised during execution, such as `ConflictExceptions` and `DeadPlaceExceptions`, and consequently restarts the execution until it succeeds or a catastrophic failure occurs.

The `Tx` class provides the functions `get` and `set` for reading and writing resilient data at the calling place. Expanding the transaction scope to other places can be done using the `asyncAt` function, which identifies the target place using its virtual id. The following code uses the transaction store APIs to perform a bank transaction by moving $100 from account A located at place 1, to account B located at place 2. Using the virtual place identifiers enable `executeTransaction` to seamlessly execute the transaction even if the physical places change during execution.

```
1 def bank(store:TxStore[String,Long]) {
2     val tx = store.makeTransaction();
3     store.executeTransaction((tx:Tx[String,Long]) => {
4         tx.asyncAt( 1, ()=> {  tx.set ("A", tx.get("A") - 100 )  });
5         tx.asyncAt( 2, ()=> {  tx.set ("B", tx.get("B") + 100 )  });
6     });
7 }
```

Asynchronous Recovery

When a `TxStore` is created, an instance of it is registered in the runtime system at every place (active and spare). Normally in RX10, the network layer notifies the runtime layer when a place fails. We modified the notification handler to not only recover the control flow by updating the finish objects, but to also notify the local instance of the transactional store of the failure. If the `TxStore` is configured to recover asynchronously and the failed place is the current place's slave (i.e. its right neighbor), the current place immediately starts the recovery protocol concurrently with other place tasks.

While the recovery protocol is executing, read and write actions at places that were not impacted by the failure proceed normally. However, actions targeted to the dead place and/or its two direct neighbors (its master place at the left and its slave place at the right) may temporarily fail until the recovery completes.

Replica Migratability

A challenging aspect of asynchronous recovery is how to handle data migration concurrently with the store's read/write actions. This challenge is avoided in the `PlaceLocalStore` by requiring the program to stop using the store while it is being recovered. Because no write actions are performed on any of the replicas during recovery, all the replicas are in a *migratable* state and can be safely copied to the spare places, as shown in Listing 5.2. However, in the `TxStore`, the replicas at the neighboring places of the dead place may not be in a migratable state immediately after the failure.

The master of the dead place can be in the middle of handling read/write transactions either as a transaction coordinator or a transaction participant. Some of these transactions may have passed the prepare phase of the 2PC protocol and are waiting for the final `COMMIT` or `ABORT` message. The master has no choice but to wait for these transactions to terminate. It cannot abort them in order to start the recovery, because the `2PC` protocol does not permit a place to abort a transaction after it has voted to commit it. For transactions that did not yet receive the `PREPARE` message, the master has the choice to abort them or allow them to terminate.

On the other hand, the slave of the dead place maintains prepare-logs for the transactions of its master that are prepared to commit. When the master dies, some of the transaction logs may be in transit to the slave place and will eventually be received by it. Unfortunately, the slave cannot independently determine whether to commit or abort these pending transactions. The slave, therefore, has no choice but to wait for these transactions to terminate by receiving a `COMMIT` or `ABORT` message from their coordinators. Because the transaction coordinator is always available (see Section 5.2.3.5), the slave will eventually receive these required messages.

The state diagrams in Figure 5.10 describe how a master or a slave place make a transition from a non-migratable state to a migratable state.

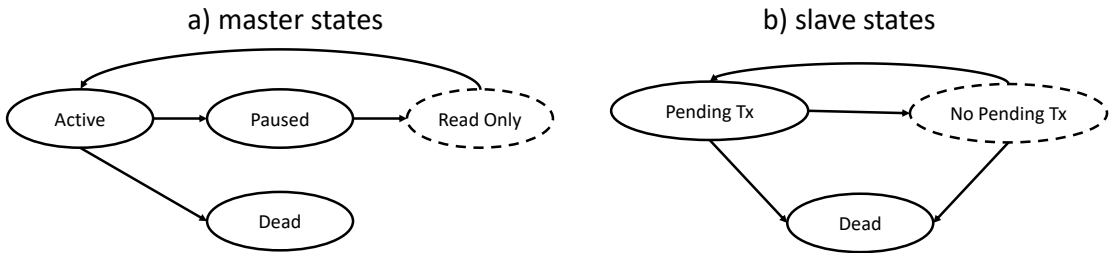

Figure 5.10: Master replica and slave replica state diagrams. A solid line means a non-migratable state, and a dotted line means a migratable state.

A master place can be in one of four states:

1. Active: the default state in which all actions are accepted normally.
2. Paused: a temporary state for preparing the master to be migratable. In this state, the master aims to finalize any transactions prepared to commit and pauses accepting write transactions temporarily. It accepts `COMMIT` and `ABORT` messages for prepared transactions and aborts any non-prepared write transactions. Read-only transactions are kept active unless they attempt to upgrate from read-only to write; in that case, they will be aborted.
3. Read-only: no prepared write transaction is pending, and only read actions are permitted on the master replica. The read-only state is the only migratable state for a master place.
4. Dead: the master has died. We assume the master can reach a dead state from the active state only, because the other states imply that the slave is also dead. A failure of both a master and its slave is unrecoverable in our system.

A slave place can be in one of three states:

1. No pending transactions: the slave does not hold prepare-logs for any transaction. It is the only migratable state for a slave place.
2. Pending transactions: the slave holds pending prepare-logs and is expecting `COMMIT/ABORT` messages regarding them from the transaction coordinators.
3. Dead: the slave has died. It is reachable from the two other states.

Note that the waiting time for a transaction to terminate is limited due to the transaction termination guarantee described in Section 5.2.3.5.

Replication Recovery Using Finish

In Listing 5.3, we outline the pseudocode for the asynchronous recovery procedure. Three places are involved in the recovery of a place: its master (left neighbor), its slave (right neighbor), and the spare place that will replace it.

The entire recovery procedure is performed using one **finish** block (Lines 7–26) that controls three tasks. The first task is sent to the slave place (Lines 9–12); it enters a busy loop waiting to reach the migratable state "no pending transactions" (Line 10), after that, it sends a copy of its data to the spare place (Line 11). The slave place will not receive new pending requests until after the spare place is fully recovered, therefore, the slave place is automatically paused. The second task is created at the master place (Lines 14–19); it starts by pausing the master replica (Line 15) to prepare it to reach the read-only state. It waits until the read-only state is reached (Line 16), before sending a copy of the master replica to the spare place (Line 17). Finally, it activates the master replica to continue processing the transactions normally (Line 18). The third task is sent to the spare place (Lines 21–25); it aims to activate the spare place after ensuring that it received the replicas successfully from the master and slave places. Since it knows their identities, it can break its busy waiting and throw a `DeadPlaceException` if it detects that either the master or the slave has died.

Listing 5.3: TxStore asynchronous recovery.

```
1 /*called by the master of the dead place*/
2 def recover(deadPlace:Place) {
3     val left = here;
4     val right = store.activePlaces().next(deadPlace);
5     assert !right.isDead();
6     val spare = store.allocateSpare();
7     finish {
8         //create the spare's master replica
9         at (right) async {
10            slave.waitUntilMigratable();
11            slave.copyToMaster(spare);
12        }
13        //create the spare's slave replica
14        at (left) async { //local task (left = here)
15            master.pause();
16            master.waitUntilMigratable();
17            master.copyToSlave(spare);
18            master.activate();
19        }
20        //activate the spare after receiving its replicas
21        at (spare) async {
22            slave.waitForReplicaFrom(left);
23            master.waitForReplicaFrom(right);
24            master.activate();
25        }
26    }
27
28    //assign work to the spare place
29    at (spare) @Uncounted async { store.work(store); }
30 }
```

After the spare place has been initialized, it can participate in the computation by performing a user-defined task. The user provides this task in the `work` closure given to the store at construction time (see the parameters of `TxStore.make` in Table 5.4). The `recover` function invokes the `work` closure asynchronously at the spare place (Line 29), only after the **`finish`** block terminates successfully.

The `@Uncounted` annotation, used in Line 29, makes the spawned **`async`** not tracked by any **`finish`**. However, the application can programmatically integrate this **`async`** with existing application tasks, for example, by exchanging certain notification messages. We will demonstrate this mechanism in the context of the parallel workers framework in Section 5.3.3.

Handshaking

When a spare place takes the role of a dead place, the other places need to recognize this change. A simple mechanism to achieve this goal is to perform a global handshaking operation from the spare to all other places. However, eagerly notifying all places not only slows down the store's recovery, but may be unnecessary for some places that do not need to communicate with that place. For this reason, we implemented an on-demand handshaking mechanism that works as follows: when a place detects the failure of another place, it queries its master for the physical identity of the dead place given its virtual id and updates its local activePlaces group accordingly. If the master of the dead place is also dead, a catastrophic failure is detected, and the entire application terminates.

Summary

The `TxStore` provides a simple programming interface for the application to store critical application data that require atomic multi-place updates. The asynchronous recovery mechanism minimizes the impact of failures to only the places in direct communication with the dead place. Transactions targeted to the dead place and only write transactions targeted to the master of the dead place are temporarily aborted. By retrying these transactions, they will eventually succeed after the dead place has been recovered. Spare places can automatically join the computation without global synchronization with other places. In Section 5.3.3, we describe the parallel workers application framework that uses the `TxStore` as its foundation for data resilience.

5.3.2 Resilient Iterative Framework

The resilient iterative framework can be used to implement fault tolerant iterative algorithms based on coordinated in-memory checkpointing. It enables the developer to focus on the algorithmic aspects of the application and rely on the framework to orchestrate the fault tolerance aspects. Meanwhile, it makes it possible for the application to optimize the performance by controlling the checkpointing scope.

We applied multi-resolution resilience by building the iterative framework based on the `PlaceLocalStore` (Section 5.3.1.1). We chose this particular store type because atomic multi-place updates are not needed and because global recovery is well-aligned with coordinated checkpointing. Table 5.5 lists the APIs of the iterative framework.

Table 5.5: The resilient iterative application framework APIs.

Class	Function	Returns
IterativeApp	isFinished()	Boolean
	step()	void
	checkpoint()	HashMap[K,V]
	restore(ckptState:HashMap[K,V])	void
	remake(changes:ChangeDescription)	Any
GlobalIterativeExecutor	make(ckptInterval:Long, spareCount:Long)	IterativeExecutor
	activePlaces()	PlaceGroup
	execute(app:IterativeApp)	void
SPMDIterativeExecutor	make(ckptInterval:Long, spareCount:Long)	IterativeExecutor
	activePlaces()	PlaceGroup
	execute(app:IterativeApp)	void

The programmer implements an iterative algorithm according to the specification of the `IterativeApp` interface:

- `isFinished()`: defines the application's convergence condition.
- `step()`: defines the logic of a single iteration.
- `checkpoint()`: inserts critical application data in a key-value map for checkpointing.
- `restore(ckptState)`: updates the program state given the last checkpointed state.
- `remake(changes)`: provides the program with information about changes impacting the places, to rebuild its distributed data structures.

The programmer implements the above methods without attention to failures. The `step` and `isFinished` functions are the same in non-resilient and resilient modes. X10 applications use global data structures mapped to a group of places. The `remake` method is provided to enable the program to reconfigure its global data structures to adapt to the new place organization.

The framework provides two classes that execute an `IterativeApp` resiliently: `GlobalIterativeExecutor` and `SPMDIterativeExecutor`. Both are initialized with a user-defined checkpointing interval and a number of spare places. The framework manages the group of active places, which the program can obtain by calling the function `activePlaces()`.

5.3.2.1 The Global Iterative Executor

The `GlobalIterativeExecutor` can be used for applications with arbitrary communication patterns. It expects the programmer to define the `step` function as a global function that internally distributes the step work across the active places in any way. Listing 5.4 outlines the execution procedure followed by the global executor:

Listing 5.4: The global executor.

```
1 def execute(app:IterativeApp) {
2     var i:Long = 1;
3     var err:Boolean = false;
4     while (true) {
5       try {
6          if (err) {
7              handlePlaceChanges(app);
8              globalRestore(app);
9              i = 1;
10             err = false;
11         }
12         if(app.isFinished()) break;
13         finish app.step(); // global step
14         i++;
15         if(i % ckptInterval == 0) globalCheckpoint(app);
16      } catch (e:Exception) {
17          if(e.isDPE()) err = true; else throw e;
18      }
19    }
20 }
```

The **finish** at Line 13 waits for the completion of all child tasks created by the `step` function and raises any detected exceptions during `step` execution to the framework. The executor aims to guard the application from failures occurring during step execution, checkpointing, or restoration as long as the failure is not catastrophic. Detected exceptions that are not of type `DeadPlaceException` are considered catastrophic and cause the executor to stop and raise the error to the program (Listing 5.4-Line 17).

Checkpointing

Using a `PlaceManager pm`, a `PlaceLocalStore rs`, and a checkpoint identifier `key`, checkpointing is performed as follows:

```
21 def globalCheckpoint(app:IterativeApp) {
22     // alternating key for double-buffering
23     val k = key.equals("red") ? "black" : "red";
24     finish (for p in pm.activePlaces()) at (p) async {
25         rs.set(k, app.checkpoint());
26     }
27     key = k;
28 }
```

Double-buffering is used to guard against failures that occur during checkpointing. We maintain the last two checkpoints of each place identified as black and red in Line 23. The color of the last valid checkpoint is stored in `key`. On creating a new checkpoint, the other color is used for checkpointing the state of all the active places (Line 24-Line 26). Only after checkpointing successfully completes, the `key` variable is modified to carry the color of the new checkpoint (Line 27). A failure during checkpointing causes the code to skip over Line 27, thereby leaving the previous checkpoint as the valid one for recovery.

Recovery

When a recoverable failure is detected in Line 17, the `err` variable is set to **`true`** to direct the execution towards the recovery path. The executor recovers the application by first calling the `handlePlaceChanges` function in Line 7. As shown in the code below, this function rebuilds the group of active places (Line 30), recovers the resilient store (Line 31 — a catastrophic error will be raised if both replicas of a place are lost), and calls the `remake` function to allow the application to reconfigure its global data structures (Line 32). However, it does not handle rolling back the application state. To recover the application state, the executor calls the `globalRestore` function outlined below. It creates a task at each place that retrieves the last checkpoint from the resilient store and feeds it to the program by calling the `restore` function (Line 36). As now both the structure of the application and its data have been recovered, execution can resume normally.

```
29 def handlePlaceChanges(app:IterativeApp) {
30     val changes = pm.rebuildActivePlaces();
31     rs.recover(changes);
32     app.remake(changes);
33 }
34 def globalRestore(app:IterativeApp) {
35     finish for(p in pm.activePlaces()) at(p) async {
36         app.restore(rs.get(key));
37     }
38 }
```

5.3.2.2 The SPMD Iterative Executor

The `SPMDIterativeExecutor` is optimized for a bulk-synchronous application, which executes as a series of synchronized steps across all the active places. A global `step` for such an application would create a fan-out finish at all the places to execute only one iteration. The repeated creation of remote micro tasks for each step would unnecessarily harm the performance. Listing 5.5 shows how the SPMD executor avoids this overhead by creating a coarse-grained task at each place that can execute multiple iterations, take periodic checkpoints, and perform part of the recovery work (Lines 14–28). Note that a program that uses the SPMD executor needs to implement the `step` function as a local function handling the work of a single place.

Listing 5.5: The SPMD executor.

```
1 def execute(app:IterativeApp) {
2     var err:Boolean = false;
3     while (true) {
4         try {
5             val res:Boolean;
6             if (err) {
7                 handlePlaceChanges(app);
8                 res = true;
9                 err = false;
10            } else {
11                res = false;
12            }
13            val team = new Team(pm.activePlaces());
14            finish for (p in pm.activePlaces()) at(p) async {
15                if (res) {
16                    app.restore(rs.get(key));
17                }
18                var i:Long = 1;
19                while (!app.isFinished())  {
20                    finish app.step(); // local step
21                    i++;
22                    if(i % ckptInterval == 0) {
23                        val k = key.equals("red") ? "black" : "red";
24                        rs.set(k, app.checkpoint());
25                        team.agree(1);
26                        key = k;
27                    }
28                }
29            }
30            break;
31        } catch (e:Exception) {
32            if(e.isDPE()) err = true; else throw e;
33        }
34    }
35 }
```

In a failure-free execution, only one fan-out finish (Line 14) will be used for the entire computation. Otherwise, when a failure occurs, all the tasks terminate, and a new fan-out finish will be created after recovering the application's structure. Recovering the application data is done by the main fan-out finish (at Line 16).

Checkpointing

When a place reaches a checkpointing iteration, it independently saves its state in the resilient store using the opposite color of the last checkpoint (Lines 23–24). Before the places switch to the new checkpoint (by updating the `key` variable in Line 26), global coordination is needed to ensure the successful creation of the new checkpoint at all the places. That is achieved using the fault tolerant collective agreement function

provided by MPI-ULFM (see Section 3.4.4.3). This function is the only collective function that provides uniform failure reporting across all places. If one place is dead, the call to `team.agree` in Line 25 will throw a `DeadPlaceException` at all the places, and the recovery steps will consequently start.

Recovery

When a global failure is reported by `team.agree`, the fan-out **`finish`** at Line 14 terminates, and the control returns to the executor's main thread. Handling a recoverable failure starts by recovering the application's structure by calling `handlePlaceChanges`. After that, a new fan-out **`finish`** starts, in which the first step to be performed at all places is recovering the application's state. That is more efficient than calling the global executor's `globalRestore` method, which creates a separate fan-out finish for data recovery only.

5.3.3 Resilient Parallel Workers Framework

The resilient parallel workers framework can be used to implement an embarrassingly parallel computation that uses a group of asynchronous workers to process global data. A worker is implemented as a place. A failed worker is asynchronously recovered by assigning its data and work to a new worker. To boost the programmer's productivity, the framework completely hides the details of failure detection, data recovery, and failed worker replacement from the programmer.

We applied multi-resolution resilience by building the parallel workers framework based on the `TxStore` (Section 5.3.1.2). We chose this particular store type because it supports asynchronous recovery and enables the workers to perform resilient transactions on global data.

Table 5.6: The parallel workers application framework APIs.

Class	Function	Returns
ParallelWorkersApp[K,V]	startWorker(store:TxStore[K,V], recover:Boolean)	void
ParallelWorkersExecutor[K,V]	make(pg:PlaceGroup, cc:String , app:ParallelWorkersApp[K,V])	ParallelWorkersExecutor[K,V]
	execute()	void
	activePlaces()	PlaceGroup

Table 5.6 lists the APIs of the parallel workers framework. To use the framework, the programmer implements the `ParallelWorkersApp` interface, which defines one function `startWorker`. This function encapsulates the work of a single worker. It takes as parameters an instance of `TxStore` and a flag to indicate whether the worker is a normal worker or a recovered worker.

To invoke the execution of the application, the programmer needs to create an instance of the framework's executor by calling `ParallelWorkersExecutor.make`. It

takes as parameters the initial group of active places, the chosen CC mechanism (see Sections 5.2.3.3), and an instance of `ParallelWorkersApp`. The executor uses these parameters to construct a transactional store by calling `TxStore.make` (see Table 5.4 for the parameters of this function). One of the parameters of `TxStore` is the recovery closure `work` that defines the work to be done by a joining spare place that is replacing a dead place. The executor defines the `work` as a call to the `executeWorker` function shown in Listing 5.6 (Lines 11–20), which we will explain next.

Listing 5.6: The parallel workers executor.

```
1 def execute() {
2     pending = store.activePlaces().size();
3     for (p in store.activePlaces()) at (p) @Uncounted async {
4         executeWorker(false);
5     }
6     //block until all workers notify
7     latch.await();
8     if (recordedErrors() != null)
9         throw recordedErrors();
10 }
11 def executeWorker(recover:Boolean) {
12     var err:Exception = null;
13     try {
14         finish app.startWork(store, recover);
15     } catch (ex:Exception) {
16         err = ex;
17     }
18     //notify work completion
19     at (executor) @Uncounted async executor().notify(err);
20 }
21 def notify(err:Exception) {
22     if (err != null) recordError(err);
23     var finished:Boolean = false;
24     atomic {
25         pending --;
26         if (pending == 0)
27             finished = true;
28     }
29     if (finished) latch.release();
30 }
```

5.3.3.1 Parallel Workers Executor

We turn now to explaining how the parallel worker executor tracks the execution of the workers, and why we could not use **`finish`** for this purpose.

To start the execution, the program calls the `execute` function shown in Listing 5.6 (Lines 1–10). This function starts an asynchronous task at each active place to execute the function `executeWorker` (Lines 11–20). The executor then waits for the completion of these tasks; however, rather than using **`finish`** to track their termination, it tracks

them manually by blocking on a `latch` (Line 7). **`finish`** can only track tasks created within its scope. If the executor surrounds the worker tasks with a **`finish`**, and then later the `TxStore` creates a recovery task at the spare place, there is no way to attach this recovery task to the scope of that **`finish`**. One can solve this problem, by disabling the `TxStore`'s recovery and making the executor itself create the recovery tasks after the original tasks complete. However, this can significantly delay the recovery because **`finish`** will not complete until all the surviving workers complete. For this reason, we avoided the use of **`finish`** and used manual tracking instead. The `@Uncounted` annotation, used in Line 3 and Line 19, makes the spawned **`async`** not tracked by any **`finish`**, which we need in this framework.

For N active places, the executor expects to receive N notification messages for the execution to complete. The function `executeWorker`, called by each worker, is designed to notify the executor after completing its work (Line 19) either successfully or unsuccessfully. Every notification decrements the number of pending workers, and when this number reaches zero, the `latch` is released (at Line 29). If one worker dies before sending its notification, the spare place that will take its place will invoke `executeWorker` and therefore send the notification. A usage example of the parallel workers framework is shown in Listing 5.7.

We used the parallel workers framework for implementing two benchmarks: `ResilientTxBench`, which is used for evaluating the throughput of the transactional store (Section 5.4.2.1), and SSCA2 kernel-4, which implements a graph clustering algorithm (Section 5.4.2.2).

Listing 5.7: A usage example of the parallel workers framework.

```
1 class PlaceWorker[String,Long] extends
2         ParallelWorkersApp[String,Long] {
3     val WORK_SIZE = 10;
4     def startWorker(store:TxStore[String,Long], recover:Boolean) {
5         //identify the scope of the worker's task
6         val indx = store.activePlaces().indexOf(here);
7         val first = indx * WORK_SIZE;
8         val end = first + WORK_SIZE;
9         //process the task and save the result in the store
10        val result = do_something(first, end);
11        store.put("result:" + indx, result);
12    }
13    static def main(args:Rail[String]) {
14        val SPARE_PLACES = 1;
15        val pm = new PlaceManager(SPARE_PLACES, false);
16        val pg = pm.activePlaces();
17        val app = new PlaceWorker[String,Long]();
18        val executor = ParallelWorkersExecutor[String,Long].make(
19            pg, "RL_EA_UL" , app);
20        executor.execute();
21    }
22 }
```

5.4 Performance Evaluation

The objective of this section is to gain insight into the throughput that transactional `finish` can deliver for different classes of applications in non-resilient and resilient modes. We also aim to evaluate the performance of transactional and non-transactional RX10 applications that use our application frameworks for adding resilience.

Section 5.4.2 focuses on transactions. Using a micro-benchmark program, called `ResilientTxBench`, we measure the resilient store's throughput with transactions of different sizes, different writing load, and using different CC mechanisms. After that, we evaluate the performance of transactions using a realistic application that performs a graph clustering algorithm.

Section 5.4.4 focuses on iterative applications with global checkpointing. We evaluate the performance of three widely-known machine learning applications (linear regression, logistic regression, and PageRank) and a scientific simulation application (LULESH).

5.4.1 Experimental Setup

We conducted the following 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. Unless otherwise specified, we configured our jobs to allocate 10 GiB of memory per node. We used core binding and mapped the places to the nodes in a round-robin manner to ensure that neighboring places are located at different nodes. Each X10 place was configured to use one immediate thread by setting the environment variable `X10_NUM_IMMEDIATE_THREADS`=1. Killing places for evaluating failure scenarios is achieved by invoking the operating system's `SIGKILL` signal at the victim places. MPI-ULFM was built from revision `e87f595` of the `master` branch of the repository at `https://bitbucket.org/icldistcomp/ulfm2.git`. Unless otherwise specified, the code of the benchmarks is available in revision `8f03771` 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.

5.4.2 Transaction Benchmarking

We start by the performance evaluation results related to the use of distributed transaction in a micro-benchmark program called `ResilientTxBench`, and in a graph clustering algorithm from the Scalable Synthetic Compact Application (SSCA) benchmark suite, number 2.

5.4.2.1 ResilientTxBench

Using the parallel workers framework, we designed the `ResilientTxBench` program as a flexible framework for evaluating the throughput of the transactional store with

different configurations. The program is located at `x10.dist/samples/txbench` in our repository. Table 5.7 lists available parameters in `ResilientTxBench` and the values used in our evaluation.

Table 5.7: ResilientTxBench parameters.

	Description	Used values
p	number of worker places	16 32 64 128
t	number of producer threads per place	4
d	duration in ms for a benchmark execution	5000
c	the concurrency control mechanism	RL_EA_UL RV_LA_WB
r	the range of keys in the store	2^{18}
u	percent of update operations	0 50
h,o	h × o is the transaction size. h is the number of participating places and o is the number of operations per participant	2×8 4×4 8×2
f	transaction coordinator is one of the participants	true

The program starts $p * t$ transaction producer threads, where p is the number of worker places and t is the number of producer threads per place. Each producer thread starts a loop that generates one transaction per iteration and terminates after d milliseconds have elapsed. According to the given transaction size, a producer thread selects h different participants and o random keys at each participant. The readset and the writeset of the transaction are randomly selected from the range of r keys of the store, based on the configured update percentage u. The keys are evenly partitioned among the places. By setting $f = true$, we request that the first participant is always the place initiating the transaction; the remaining $h - 1$ participants are selected randomly. We believe this strategy is well aligned with the PGAS paradigm, as users are expected to favor locality by initiating transactions from one of the participating places.

Each producer thread records the number of completed transactions T_{ij} and the exact elapsed time in milliseconds E_{ij}, where i is the place id and j is the thread id within that place. On completion, the first place collects these values from each thread and estimates the throughput *(operations per millisecond)* as:

$$\text{throughput} = \frac{\sum_{i=1}^{p}\sum_{j=1}^{t} T_{ij} * h * o}{\sum_{i=1}^{p}\sum_{j=1}^{t} E_{ij}} \times p \times t$$

Although we implemented the six CC mechanisms described in Section 5.2.3.3, we focus our evaluation on only two of these mechanisms: RL_EA_UL and RV_LA_WB, which, respectively, represent the two extremes of pessimism and optimism in lock-based concurrency control. RL_EA_UL acquires the read locks and write locks before accessing any key and keeps acquiring the locks until the transaction terminates. On the other hand, RV_LA_WB does not acquire any locks until the commit preparation phase, which provides more concurrency to the application. In non-resilient mode, RL_EA_UL has an advantage over RV_LA_WB, since the former does not require a preparation phase for non-resilient transaction [Bocchino et al., 2008]. In resilient mode, RL_EA_UL can only avoid the preparation phase in read-only transactions.

We started four producer threads per place and allocated two cores per producer thread. Therefore, each X10 place was configured with 8 worker threads using `X10_NTHREADS`=8. We found that this level of parallelism is necessary for achieving scalable performance, because it provides enough capacity to a place to coordinate its own transactions and participate in transactions originated at other places. We measured the throughput with increasing numbers of producer threads: 64, 128, 256, and 512, and we expect the ideal performance to result in an increase in the throughput that is proportional to the number of worker threads. We evaluated the transactional store throughput in non-resilient mode and in resilient mode using the two implementations of the optimistic finish protocol: optimistic place-zero (O-p0) and optimistic distributed (O-dist).

Figure 5.11 shows the throughput results for read-only transactions, and Figure 5.12 shows the throughput results for transactions performing 50% update operations. For each configuration, we report the median, the 25th and the 75th percentile. Table 5.8 shows a summary of the results with 512 producer threads.

From these results, we draw the following conclusions:

- The centralized optimistic finish implementation scales very poorly in this benchmark, due to the large number of concurrent transactions. The distributed implementation achieves much better scaling performance for this benchmark.

- Read-only transactions scale well in both non-resilient and resilient (O-dist) modes with both CC mechanisms.

- The scalability with write transactions varies depending on the CC mechanism used. While RV_LA_WB scales reasonably well, RL_EA_UL struggles to scale as the number of threads increases.

- With read-only transactions, RL_EA_UL enables the program to achieve higher throughput than with RV_LA_WB. For example, with 512 threads, RL_EA_UL achieves an increase in throughput between 16% to 33% compared to RV_LA_WB. However, the situation is reversed with write transactions.

- With write transactions, RV_LA_WB enables the program to achieve higher throughput than with RL_EA_UL. For example, with 512 threads, RV_LA_WB

achieves an increase in throughput between 80% to 222% throughput increase compared to RL_EA_UL.

- With 512 threads, the throughput loss due to the resilience overhead of the X10 runtime and the resilient two-phase commit protocol ranges between 40%–49% for read-only transactions and between 41%–65% for write transactions. The overhead increases as the number of transaction participants increases.

Bocchino et al. [2008] performed comparative evaluation between the six CC mechanisms for a non-resilient DTM system for the Chapel language. They found that RL_EA_* mechanisms were always superior to other mechanisms because they avoid the commit preparation phase. On the other hand, our results demonstrate that the relative efficiency of different CC mechanisms depends on the application worload. Write-intensive applications can benefit more from an optimistic concurrency control mechanism where locking is delayed to the commitment time. The performance of the SSCA2-k4 clustering application, which we will explain in the next section, confirms this finding and provides more detailed analysis. Our work also demonstrates that RL_EA_* cannot always avoid the preparation phase when used in resilient mode.

There are many implementation differences that prevent direct comparison with the work in Bocchino et al. [2008]. For example, they propose a low-level implementation at the memory manager level, while our implementation is done at the runtime and application level. While we use a read-write wait-die lock, they use an exclusive write lock. Also, fault tolerance is not among their design goals.

We believe the achieved resilience overhead is acceptable, given the underlying tracking and commitment mechanisms on replicated data. We expect the use of small-sized transactions in compute-intensive applications to result in minimal resilience overhead in realistic application scenarios. The SSCA2-k4 application, which we will explain next, does not fit in this category; it is a communication intensive application that relies heavily on distributed transactions.

Table 5.8: ResilientTxBench transaction throughput (operations per ms) with 512 threads, in non-resilient and resilient mode with O-dist. The resilience overhead is shown in parentheses.

	2x8		4x4		8x2	
	non-res.	O-dist	non-res.	O-dist	non-res.	O-dist
RL_EA_UL (u=0%)	7678	4278 (44%)	3951	2129 (46%)	2077	1064 (49%)
RV_LA_WB (u=0%)	5776	3464 (40%)	2974	1714 (42%)	1570	914 (42%)
RL_EA_UL (u=50%)	1082	533 (51%)	628	341 (46%)	489	214 (56%)
RV_LA_WB (u=50%)	2893	1715 (41%)	1669	960 (42%)	1085	385 (65%)

5.4.2.2 Graph Clustering: SSCA2 Kernel-4

The SSCA benchmark suite is used for studying the performance of computations on graphs with arbitrary structures and arbitrary data access patterns. In order

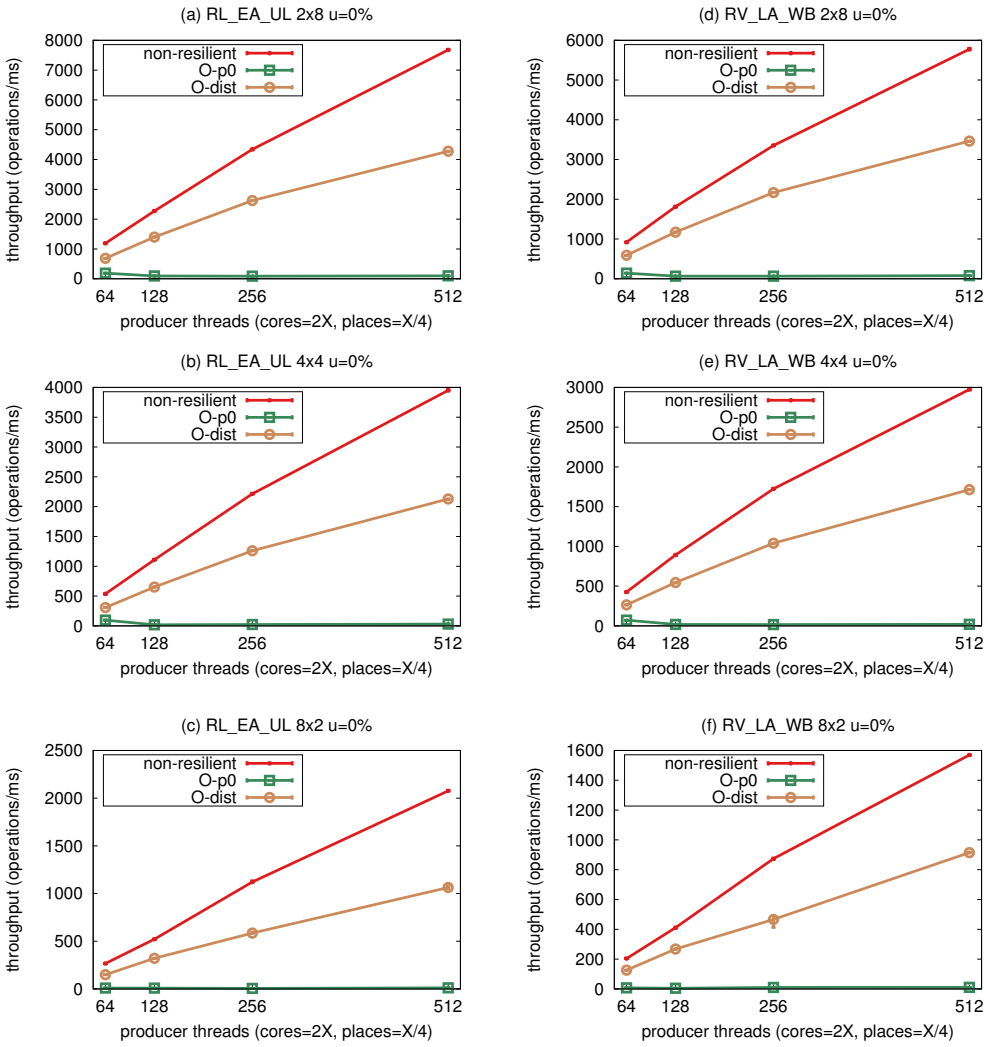

Figure 5.11: ResilientTxBench transaction throughput with 0% update.

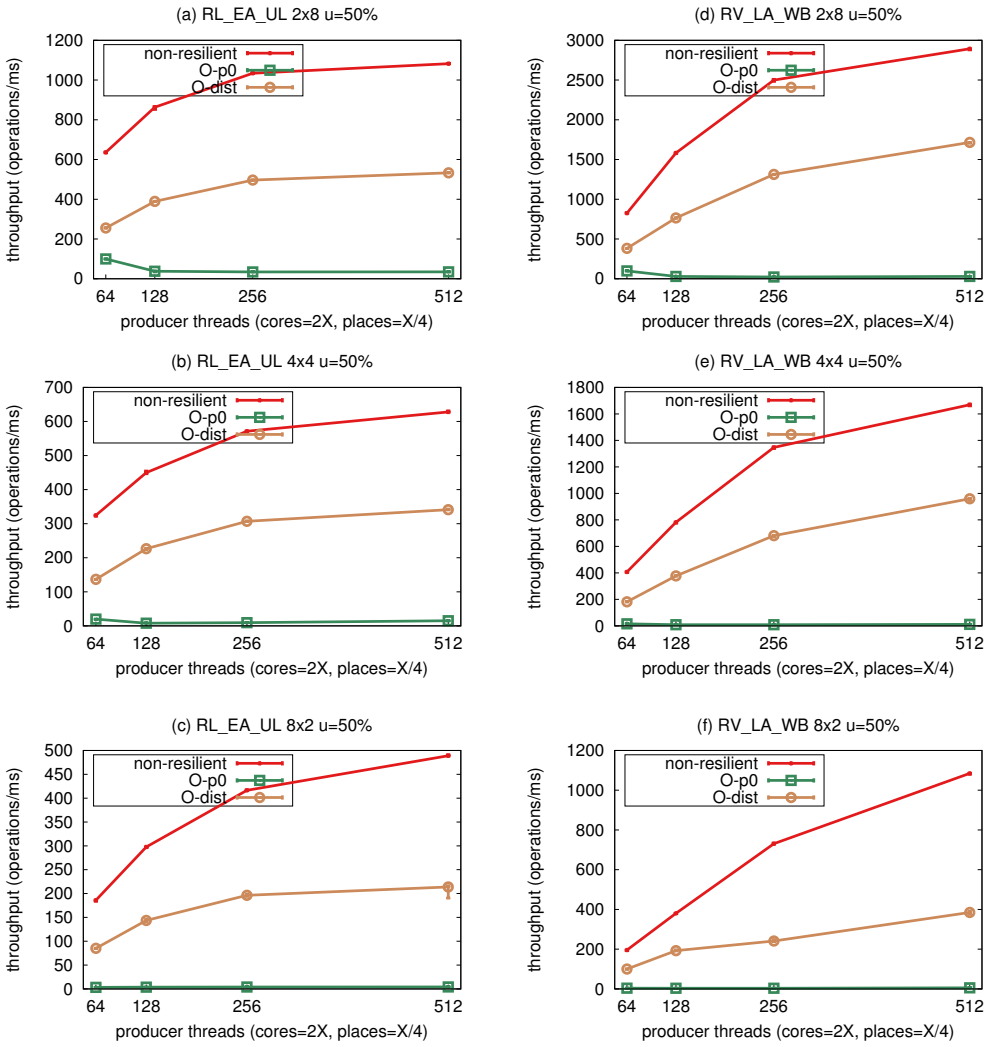

Figure 5.12: ResilientTxBench transaction throughput with 50% update.

to evaluate the performance of distributed transactions in a realistic application, we implemented kernel-4 of the SSCA benchmark number 2 [Bader and Madduri, 2005]. SSCA2-k4 is a graph clustering problem that aims to create clusters of a certain size within a graph. For graph generation, we used the SSCA2 graph generator provided in X10's benchmarking suite [X10 Benchmarks], located at `x10-benchmarks/PERCS/SSCA2/Rmat.x10`. The graph generation algorithm is based on the R-Mat model [Chakrabarti et al., 2004], which uses four parameters to configure the structure of the graph (a, b, c, and d) such that $a + b + c + d = 1$. Each of these parameters represents the probability that a vertex will be located in a specific area in the graph. We partitioned the graph vertices evenly among the places; however, the graph structure itself is replicated at all places to speed up neighbor-vertex queries, which are extensively used in this benchmark.

Using the parallel workers framework, we developed a lock-based implementation and a transactional implementation of SSCA2-k4. The source code of these implementations is available in `x10/x10.dist/samples/ssca2`. The two implementations process the graph using $N * t$ parallel threads, where N is the number of places, and t is the number of clustering threads in each place. Each thread is assigned a partition of the graph vertices, where a vertex represents a local potential root for a new cluster. A thread tries to create a cluster by first allocating the root vertex, then allocating all neighboring vertices of that root that are not already allocated. After allocating the neighbors, a heuristic is applied to pick one of them and use it for further expanding the cluster by allocating the neighbors of the picked vertex. This process continues until the cluster reaches a target size, or no more vertices are available for allocation. In that case, the thread switches to the next root vertex to create a new cluster.

The lock-based implementation uses non-blocking `tryLockRead` and `tryLockWrite` functions provided by the same lock class used by transactions (see Section 5.2.3.2 for a description of the lock specification). The clustering thread first attempts to read-lock a target vertex to check if it is part of a cluster. If locking is successful and the vertex is free, it attempts to lock the vertex for writing. If locking is successful and the vertex is still free, it marks the vertex with its cluster id and moves on to another vertex. All acquired vertices remain exclusively locked until cluster creation completes. Failure to acquire a lock for read or write forces the thread to clear all allocated vertices, unlock the acquired locks, and restart processing the same root vertex. Similar to [Bocchino et al., 2008], our lock-based implementation does not protect against livelocks. The transactional implementation avoids explicit locking by handling each cluster creation as a transaction. The runtime system implicitly checks for conflicts and performs the consequent rollback of obtained vertices.

The lock-based implementation is not resilient; however, we use it as an indicative baseline for comparison with non-resilient transactions. Although adding resilience to the lock-based version is possible, we found that the implementation would be complex as it would require handling data replication and lock tracking at the application level. With transactions, however, we receive these features transparently by the runtime system. In the transaction implementation, we don't checkpoint the progress of each thread. Therefore, the threads of a recovering worker restart process-

ing their assigned vertices from the beginning. However, because the root vertices of the clusters are locally located, checking the status of previously locked vertex is not expensive as it generates a local read-only transaction.

We performed our experiment on a balanced graph of size 2^{18} vertices, with equal probability for the four graph parameters ($a = 0.25$, $b = 0.25$, $c = 0.25$, $d = 0.25$), and a target cluster size of 10. We started four clustering threads at each worker (place) and allocated two cores per clustering thread, for the same reason described in the `ResilientTxBench` experiment (Section 5.4.2.1). Therefore, each X10 place was configured with 8 worker threads using `X10_NTHREADS`=8. Experiments running in resilient mode use the optimistic distributed finish implementation (O-dist).

We report the median of five measurements for each configuration scenario for this application. The 95% confidence interval of the reported processing time is less than 8% of the mean for experiments running in non-resilient mode, less than 4% of the mean for experiments running in resilient mode without failure, and less than 14% of the mean for experiments running in resilient mode with failure.

Table 5.9 compares the performance of the application using different concurrency control mechanisms. Because this application is a write-intensive application, the lock-based implementation and RL_EA_UL suffer significantly more conflicts than RV_LA_WB. Assuming the conflict cost is the ratio of processing time to the number of conflicts: in non-resilient mode with 64 threads, the conflict cost is 0.08, 0.57, and 6.93 for lock-based, RL_EA_UL, and RV_LA_WB, respectively. The conflict cost in RV_LA_WB is much higher because conflict detection is delayed to the commit preparation time, after the transaction has fully expanded. The other two mechanisms detect conflicts much faster, but the negative consequence of their low conflict cost is retrying the acquisition of the same lock within a short time interval, which results in more repeated conflicts than RV_LA_WB. Overall, in non-resilient mode, the number of conflicts offsets the conflict cost for the two transactional implementations, and their performance is comparable. Locking outperforms the transactional mechanisms with small numbers of threads. However, this advantage is gradually lost as the number of threads increases.

In resilient mode, the cost of the transaction resilience overhead and the large gap in the total number of processed transactions increase the performance gap between RL_EA_UL and RV_LA_WB. Overall, RV_LA_WB achieves better scalability and lower resilience overhead than RL_EA_UL in this application. With 64 threads, the resilience overhead is 117% and 141% for RV_LA_WB and RL_EA_UL, respectively. With 512 threads, the overhead of RV_LA_WB reduces to only 75%, while it increases to 242% with RL_EA_UL due to the massive increase in the number of conflicts.

Using the transactional implementation, we performed an experiment using RV_LA_WB to evaluate the overhead of failure recovery for this application. We configured the middle place as a victim and forced its first clustering thread to kill the process after completing half of its assigned vertices. Table 5.10 shows the impact of the failure on the total processing time and the number of conflicts. It also shows the time consumed for replication recovery by the resilient store. As expected, the failure results in generating more conflicts by the transactions that interact with the

Table 5.9: SSCA2-k4 performance with different concurrency control mechanisms.

		Conflicts				Processing time			
	threads	**64**	**128**	**256**	**512**	**64**	**128**	**256**	**512**
Non-res.	lock-based	82831	147793	310540	986013	6.3s	4.2s	3.4s	3.6s
	RL_EA_UL	15848	42284	70933	207199	9.1s	5.2s	3.3s	3.1s
	RV_LA_WB	1414	2557	5881	13803	9.8s	5.0s	3.4s	3.6s
Res. (O-dist)	RL_EA_UL	22924	50858	186940	592674	21.9s	12.7s	11.6s	10.6s
	RV_LA_WB	1127	2382	5543	12745	21.3s	10.9s	8.7s	6.3s

dead place. The number of conflicts is proportional to the replication recovery time. With small numbers of places (16 and 32), the recovery time ranged between 1.2–3.4 seconds, and with larger numbers of places (64 and 128), the recovery time ranged between 0.3–0.4 seconds. In this strong scaling experiment, as the number of places increases, the data assigned to each place shrinks and the transfer time of replicas reduces. However, there are factors other than the replica size that can also impact the recovery performance and cause high performance variability (the 95% confidence interval of the processing time under failure is between 5%–14% of the mean). These factors include the possible delay in allocating the spare place due to high processing load at the master of the dead place and the wait time experienced by the master and the slave replicas to reach a migratable state (see Section 5.3.1.2). We expect the load at the master place to be a determining factor for the recovery performance due to performing the recovery process asynchronously with normal application tasks.

To summarize, as shown in Figure 5.13 our transactional finish implementation results in good strong-scaling performance for the SSCA2-k4 application in failure-free and failure scenarios. The performance is comparable to the lock-based implementation in non-resilient mode. We counted the number of lines of the code that performs the actual cluster generation function in both implementations, excluding the main function, definition of classes, debug messages, and code related to killing the victim places. The lock-based implementation takes 171 lines of code without any fault tolerance support, while the resilient transactional implementation uses only 80 lines of code[2].

5.4.3 Iterative Applications Benchmarking

In this section, we evaluate the performance of a suite of applications that we enhanced with resilience support using the iterative application framework. As described in Section 5.3.2, the underlying fault tolerance mechanism of this framework is periodic global checkpointing based on the `PlaceLocalStore`. We performed weak scaling experiments using 128, 256, 512, and 1024 places, with one core per place (except for LULESH, which uses 343, 512, 729, and 1000 places). Each place is config-

[2]The lines of code was counted using David A. Wheeler's 'SLOCCount'

Table 5.10: SSCA2-k4 strong scaling performance with RV_LA_WB under failure.

	threads	**64**	**128**	**256**	**512**
Failure free	Processing time	21.3s	10.9s	8.7s	6.3s
	Conflicts	1127	2382	5543	12745
One failure	Processing time	23.4s	15.6s	10.8s	7.3s
	Replication recovery time	1.2s	3.4s	0.4s	0.3s
	Conflicts	3500	5786	7021	13861

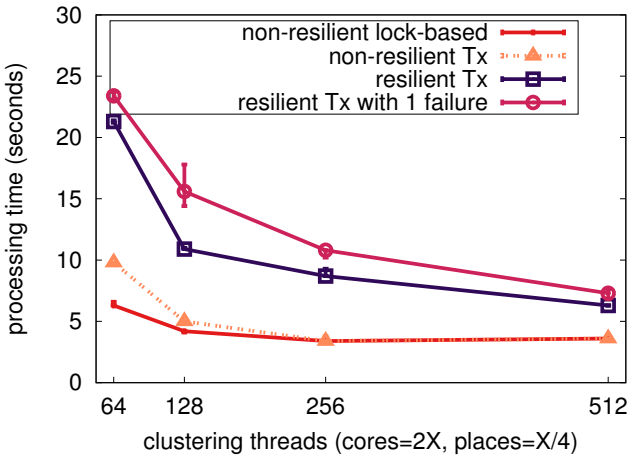

Figure 5.13: SSCA2-k4 strong scaling performance with RV_LA_WB under failure using O-dist finish.

ured to start one worker thread, using `X10_NTHREADS`=1, and one immediate thread, using `X10_NUM_IMMEDIATE_THREADS`=1. We allocated 60 GiB of memory per *Raijin* node to have sufficient space for the application data and the in-memory checkpoints.

All the applications were configured to execute 60 iterations. When resilience is enabled, checkpointing is performed every 10 iterations, with the first checkpoint taken at iteration 0. To evaluate the failure recovery performance, we simulated three failures during the execution of each application. The victim places are $N/4$, $N/2$, and $3N/4$, where N is the number of places. We forced a place failure every 20 iterations, exactly at iterations 5, 35, and 55. Therefore, a recovered application executes a total of 75 iterations, because each failure causes the application to re-execute 5 iterations. In our experiments, we measure different sources of resilience overhead impacting the applications, including the resilient **`finish`** overhead, the checkpointing overhead, and the recovery overhead.

The resilient **`finish`** overhead depends on the termination detection implementation used by the runtime system. In Chapter 4, we demonstrated the superior performance of our proposed optimistic finish protocol compared to RX10's pessimistic finish protocol using micro-benchmarks that represent different computation patterns (see Section 4.11). In addition to micro-benchmarks, we also aim to evaluate the performance of optimistic finish in realistic applications. Therefore, one of the objectives of our experiments in this section is to compare the resilience overhead of the applications using the four resilient finish implementations: O-p0, O-dist, P-p0, and P-dist. By default, the distributed implementations avoid replicating any **`finish`** originating from place-zero, thereby avoiding the cost of synchronously replicating the task signals. Because currently the failure of place-zero is unrecordable in RX10, replicating the state of these finishes is useless. However, we can disable this optimization to measure the performance when all the **`finish`** constructs are replicated. We use the modes **O-dist-slow** and **P-dist-slow** to refer to the configuration when all the **`finish`** constructs, including those hosted at place zero, are replicated.

5.4.3.1 X10 Global Matrix Library

The X10 Global Matrix Library (GML) provides a rich set of matrix formats and routines for implementing distributed linear algebra operations. It includes a suite of benchmarking applications performing common data analytic and machine learning algorithms. To simplify the development of resilient GML applications, we extended key matrix classes with snapshot and restore methods that make checkpointing and recovering easier to implement in the application [Hamouda et al., 2015]. Most of the distributed matrix operations in GML rely on collective communication between places. Initially, GML contained a custom implementation of collectives. We replaced these custom collective operations with calls to the collective operations of the `Team` class. This is done to exploit the performance and resilience capabilities provided by the `Team` class, including the use of fault tolerant collective functions from MPI-ULFM (see Section 3.4.4). Our enhancements have been contributed to the X10 code base, and they are currently available in the X10 release v2.6.1.

In this section, we evaluate the cost of resilience for three GML applications: Linear Regression, Logistic Regression, and PageRank. These applications follow the SPMD bulk-synchronous model; therefore, we enhanced them with resilience support using the SPMD iterative executor described in Section 5.3.2.2. The code of the three applications is available as part of X10 release v2.6.1 and can be located at `x10/x10.gml/examples`.

The initialization procedure of the three applications is done using a fan-out **`finish`** that instructs the places to initialize their partition of the input data. After initialization, the SPMD executor also uses a fan-out **`finish`** to supervise the execution of the steps. The three applications execute a series of coordinated steps that perform communication between places only through collective operations. Therefore, the used communication patterns in these applications is limited to fan-out **`finish`** and collective operations.

The `BenchMicro` results of the fan-out finish pattern show that the centralized implementations (O-p0 and P-p0) are more efficient for this pattern than the distributed implementations (see Section 4.11.2.2), because this pattern requires resilient tracking of only one **`finish`**. Luckily, all the applications evaluated in this chapter start their fan-out finishes from place-zero, which is expected in most X10 applications. Because replicating place-zero finishes is avoided by the O-dist and P-dist modes, their performance is expected to be similar to the performance of the centralized modes. On the other hand, the O-dist-slow and P-dist-slow are expected to suffer higher resilience overhead.

5.4.3.2 Linear Regression

The Linear Regression (LinReg) benchmark trains a linear regression model against a set of labeled training examples. We trained a model of 100 features with a training set size of 1,000,000 examples per place. The training data is generated randomly using a certain seed and partitioned evenly among the places. We configured our experiments to exclude the input training data from the checkpointing state. When a failure occurs, the active places collectively regenerate the input data using the same random seed. This avoids the high cost of serializing and checkpointing the large input matrix. Approximately 86 lines of codes out of 414 total lines were added or modified from the original implementation to conform with the `IterativeApp` interface.

Figure 5.14 shows a detailed break-down of the execution time (in seconds) with different numbers of places. Table 5.11 focuses on the results of the smallest and largest problem sizes (with 128 and 1024 places, respectively) and focuses on the O-dist mode, which achieves either the same or better performance than the other resilient modes in most of the results in this chapter. The resilient execution times mentioned in the following discussion are for the O-dist mode only.

The execution time of a single step is almost the same in non-resilient and resilient modes. Since we are using the MPI-ULFM backend, the `Team` class maps its collective operations to efficient fault tolerant MPI collectives and entirely avoids the resilience

overhead of **finish**. The resilience cost of the fan-out finish used during initialization is only measurable in the O-dist-slow and P-dist-slow modes, where the optimistic finish is about 2 seconds faster with 1024 places.

LinReg has a modest checkpointing state per place that includes only four dense vectors of 100 elements. The dense vector format enables efficient data serialization while checkpointing and restoring the data. The resulting overhead of in-memory checkpointing using the `PlaceLocalStore` is negligible for this application (only 1 ms with 1024 places). Using `MPI_COMM_AGREE` to guarantee the successful generation of a checkpoint at all places costs LinReg only 15 ms with 1024 places. That is mainly because the places take very short time to checkpoint their state. Places that take a long time for serializing and checkpointing their state delay the termination of the agreement protocol and cause the application to suffer longer checkpoint agreement times. The results of the LogReg and PageRank applications demonstrate this effect. Because LinReg checkpointing is fast, a resilient failure-free execution with six checkpoints experiences less than 0.2% resilience overhead.

When a failure occurs, the first place that detects the failure implicitly invalidates the `Team` communicator by calling `MPI_COMM_REVOKE`. The invalidated communicator causes the places to fail while executing their next collective function. The fan-out **finish** used by the iterative executor will report the failure to the executor only after all the places stop processing and raise errors. With 1024 places, the cost of failure detection and recovery is as follows. Failure detection costs LinReg about 662 ms. Replacing the failed `Team` with a new one for recovery takes about 200 ms in the three GML applications. Recovering the resilient store takes only 17 ms, thanks to the limited checkpoint state and the simplicity of dense vector serialization. The major cost for application recovery results from the `app.remake()` method. This method reorganizes the structure of the input matrix and other vectors to map to the new group of active places and reinitializes the input data at all places[3]. As a result, it costs the application about 2 seconds. Restoring the last checkpoint state at each place results in unmeasurable performance overhead. This is because each place restores its state from its local master replica, and because the checkpointed state is very small for this application. The total time for recovering the application from one failure, including the time to re-execute the last 5 steps, is about 6 seconds (15% of the total execution time in non-resilient mode).

5.4.3.3 Logistic Regression

The Logistic Regression (LogReg) benchmark trains a binary classification model against a set of labeled training examples. We trained a model of 100 features with a training set size of 1,000,000 examples per place. The training data is generated randomly using a certain seed and partitioned evenly among the places. Like LinReg, the input data is excluded from the checkpoint state and is regenerated using the same random seed for failure recovery. Approximately 110 lines of codes out of 585

[3]It is possible to optimize this method to initialize the input data only at the new added places; however, the version used in the evaluation does not perform this optimization.

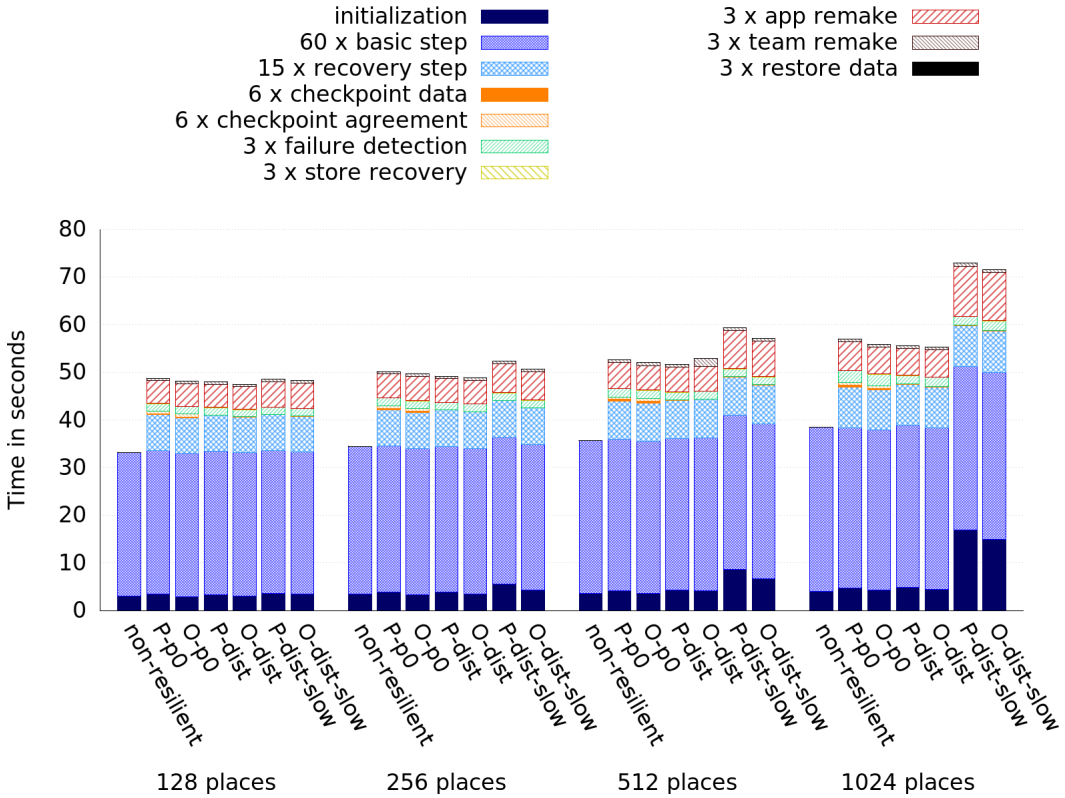

Figure 5.14: Linear regression weak scaling performance (1 core per place).

Table 5.11: Linear regression performance.

		128 places	1024 places
Non-res.	initialization	3016 ms	4042 ms
	step	503 ms	573 ms
	total	33176 ms	38422 ms
Res. (O-dist)	initialization	3038 ms	4369 ms
	step	501 ms	567 ms
	checkpoint data (1 ckpt)	1 ms	1 ms
	checkpoint agree (1 ckpt)	5 ms	15 ms
	failure detection (1 failure)	512 ms	662 ms
	store recovery (1 failure)	17 ms	17 ms
	application remake (1 failure)	1583 ms	1892 ms
	team remake (1 failure)	168 ms	206 ms
	restore data (1 failure)	0 ms	0 ms
	total (6 ckpts, 3 failures)	47489 ms	55321 ms

total lines were added or modified from the original implementation to conform with the `IterativeApp` interface.

LogReg performance results are shown in Figure 5.15 and Table 5.12. The resilient execution times mentioned in the following discussion are for the O-dist mode only.

As in LinReg, resilient execution of LogReg steps incurs no resilience overhead, due to using MPI-ULFM's collectives. The O-dist-slow and P-dist-slow cause high resilience overhead while initializing and reinitializing the application for recovery. The other resilient modes have negligible resilience overhead.

LogReg checkpoint at each place includes a large vector with 1,000,000 elements and two small vectors with 100 elements. The vectors are stored in a dense format as in LinReg. With 1024 places, LogReg takes 76 ms for saving the data in the resilient store and takes 46 ms for checkpoint agreement. The resulting resilience overhead for a failure-free execution with six checkpoints is less than 2% with 1024 places.

The resilient executor detects the failure of a place in about 705 ms. Because the checkpoint state is larger than LinReg, recovering the resilient store by transferring the checkpointed data to a new spare place is more expensive in LogReg, at 204 ms. Similar to LinReg, `Team` recovery takes about 200 ms. The most expensive part of failure recovery is remaking the application, which takes about 3 seconds in LogReg. Restoring the application state using the last checkpoint takes 26 ms. The total time for recovering the application from one failure, including the time to re-execute the last 5 steps, is about 11 seconds (12% of the total execution time in non-resilient mode).

Table 5.12: Logistic regression performance.

		128 places	1024 places
Non-res.	initialization	6561 ms	8272 ms
	step	1194 ms	1365 ms
	total	78201 ms	90192 ms
Res. (O-dist)	initialization	6649 ms	8875 ms
	step	1179 ms	1341 ms
	checkpoint data (1 ckpt)	75 ms	76 ms
	checkpoint agree (1 ckpt)	26 ms	46 ms
	failure detection (1 failure)	535 ms	705 ms
	store recovery (1 failure)	202 ms	204 ms
	application remake (1 failure)	2437 ms	2918 ms
	team remake (1 failure)	179 ms	206 ms
	restore data (1 failure)	24 ms	26 ms
	total (6 ckpts, 3 failures)	105811 ms	122359 ms

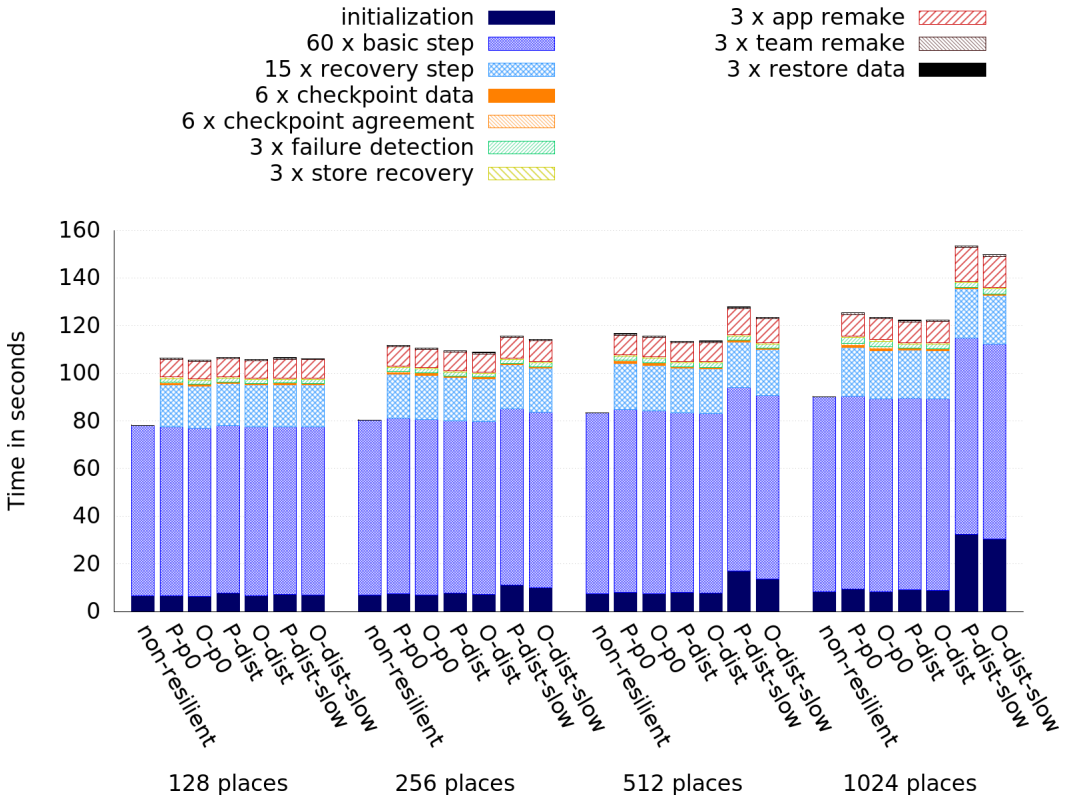

Figure 5.15: Logistic regression weak scaling performance (1 core per place).

5.4.3.4 PageRank

The PageRank benchmark computes a measure of centrality for each vertex in a graph. Our experiments use a randomly generated graph of 10 million vertices per place stored in a sparse matrix with a density of 0.001. Similar to LinReg and LogReg, we do not checkpoint the input graph but rather recompute it during failure recovery. Approximately 65 lines of codes out of 383 total lines were added or modified from the original implementation to conform with the `IterativeApp` interface.

PageRank performance results are shown in Figure 5.16 and Table 5.13. The resilient execution times mentioned in the following discussion are for the O-dist mode only.

The scaling pattern of PageRank is less efficient than LinReg and LogReg. We believe that this is due to inefficient representation of sparse matrices in the GML library. A sparse matrix in GML is based on a deep hierarchy of nested objects that complicates processing the matrix and may result in inefficient memory management overhead. Because PageRank steps use collective operations only, we expect the step execution time to be similar in non-resilient and resilient modes. However, we observed a resilience overhead between 10%–27% in experiments that used 128, 256, and 512 places. The root cause of this overhead is unclear to us at this stage. Because this problem does not occur in the applications that use dense matrix format, we aim to investigate in the future whether inefficiencies in sparse matrix representation and processing can influence the performance of RX10 applications. With 1024 places, PageRank steps do not incur any resilience overhead, as expected.

With 1024 places, PageRank checkpoint state at each place includes a large vector with about 3,000,000 elements. PageRank takes about 324 ms for saving this vector in the resilient store. Because PageRank checkpointing is more expensive than LinReg and LogReg, the checkpoint agreement time is also more expensive at 236 ms. The resulting resilience overhead for a failure-free execution with six checkpoints is less than 9% with 1024 places.

The resilient executor detects the failure of a place in about 380 ms. Because the checkpoint state is larger than LogReg, recovering the resilient store is also more expensive in PageRank at 445 ms. Similar to LinReg and LogReg, `Team` recovery takes about 200 ms. Reinitializing the application using the `app.remake` method takes about 2 seconds. Restoring the application state using the last checkpoint takes 68 ms. The total time for recovering the application from one failure, including the time to re-execute the last 5 steps, is about 6 seconds (16% of the total execution time in non-resilient mode).

5.4.3.5 LULESH

X10 provides an implementation of the LULESH proxy application [Karlin et al., 2013], which simulates shock hydrodynamics through a series of time steps. Each step executes a series of stencil operations on a block of elements and nodes that define these elements. The elements and their nodes are evenly partitioned among the places. Therefore, exchanging ghost regions between neighboring places is required for the

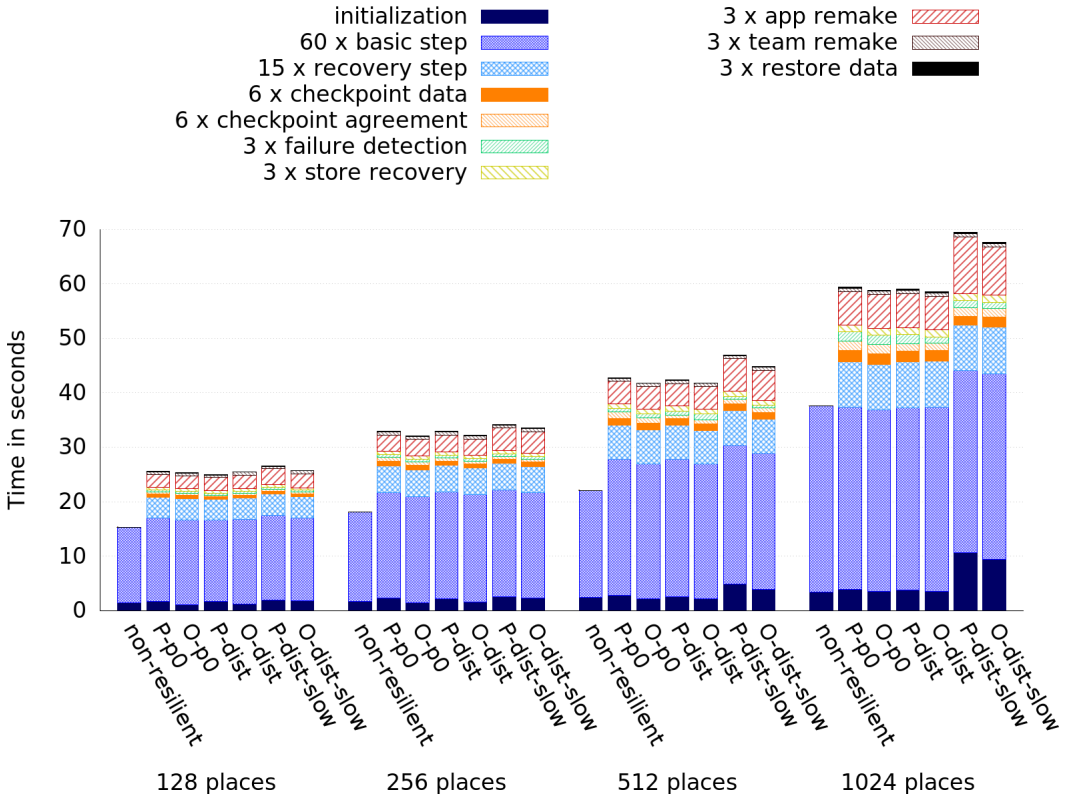

Figure 5.16: PageRank weak scaling with number of places (1 core per place).

Table 5.13: PageRank performance.

		128 places	1024 places
Non-res.	initialization	1384 ms	3385 ms
	step	232 ms	570 ms
	total	15324 ms	37565 ms
Res. (O-dist)	initialization	1237 ms	3546 ms
	step	259 ms	563 ms
	checkpoint data (1 ckpt)	89 ms	324 ms
	checkpoint agree (1 ckpt)	53 ms	236 ms
	failure detection (1 failure)	129 ms	380 ms
	store recovery (1 failure)	165 ms	445 ms
	application remake (1 failure)	831 ms	2029 ms
	team remake (1 failure)	183 ms	201 ms
	restore data (1 failure)	19 ms	68 ms
	total (6 ckpts, 3 failures)	25495 ms	58500 ms

stencil computations at each step. Reduction and barrier collective operations are also invoked by LULESH steps for synchronization and evaluating the convergence state. We enhanced LULESH with resilience support using the SPMD iterative executor. Our resilient implementation has been contributed to the X10 applications repository [X10 Applications] in the `lulesh2_resilient` folder. Approximately 200 lines of codes out of 4100 total lines were added or modified from the original implementation to conform with the `IterativeApp` interface.

We evaluate the performance of LULESH with a problem size of 30^3 elements per place. The number of places is 343, 512, 729, and 1000, because LULESH requires a perfect cube number of places. The performance results are shown in Figure 5.17 and Table 5.14.

LULESH uses a communication-intensive initialization kernel that generates a large number of concurrent **`finish`** objects and remote tasks. Each place pre-allocates buffers for holding the ghost regions and communicates with 26 neighboring places using remote **`async`**s for exchanging references to these buffers. When places die, some of these references will dangle and require updating. On recovering the application, the initialization kernel is re-executed at all places for updating the buffer references. Using an efficient termination detection protocol is crucial for achieving acceptable performance for LULESH initialization and recovery kernels. The initialization kernel incurs the following resilience overhead with 1000 places: 482% for P-p0, 335% for O-p0, 45% for P-dist, and only 40% for O-dist. When tracking place-zero finishes, the overhead is: 475% for P-dist-slow and 377% for O-dist-slow.

Unlike the GML applications, LULESH steps are subject to the **`finish`** resilience overhead as they generate pair-wise remote tasks for exchanging ghost regions. The measured resilience overhead of a single step with 1000 places is: 13% for P-p0, 8% for O-p0, 10% for P-dist, and only 4% for O-dist. The optimistic protocol is successfully reducing the resilience overhead for LULESH during initialization and step execution. As initializing the places and exchanging the ghost regions execute in parallel at all places, the distributed implementations outperform the centralized implementations in this application.

The following analysis focuses on the O-dist performance results shown in Table 5.14. LULESH has a modest checkpointing state that takes 24 ms for saving in the resilient store and 35 ms for checkpoint agreement with 1000 places. The resulting resilience overhead for a failure-free execution with six checkpoints is less than 19% (the high percentage is due to the small execution time of LULESH steps, compared to the GML applications).

The failure recovery performance with 1000 places is as follows. The resilient executor detects the failure of a place in about 364 ms. Recovering the resilient store takes only 60 ms because the checkpointed state is limited in size. Similar to the GML applications, `Team` recovery takes about 200 ms. The most expensive part of failure recovery is reinitializing the application, which takes about 2 seconds. Restoring the application state using the last checkpoint takes only 3 ms. The total time for recovering the application from one failure, including the time to re-execute the last 5 steps, is about 3 seconds (50% of the total execution time in non-resilient mode).

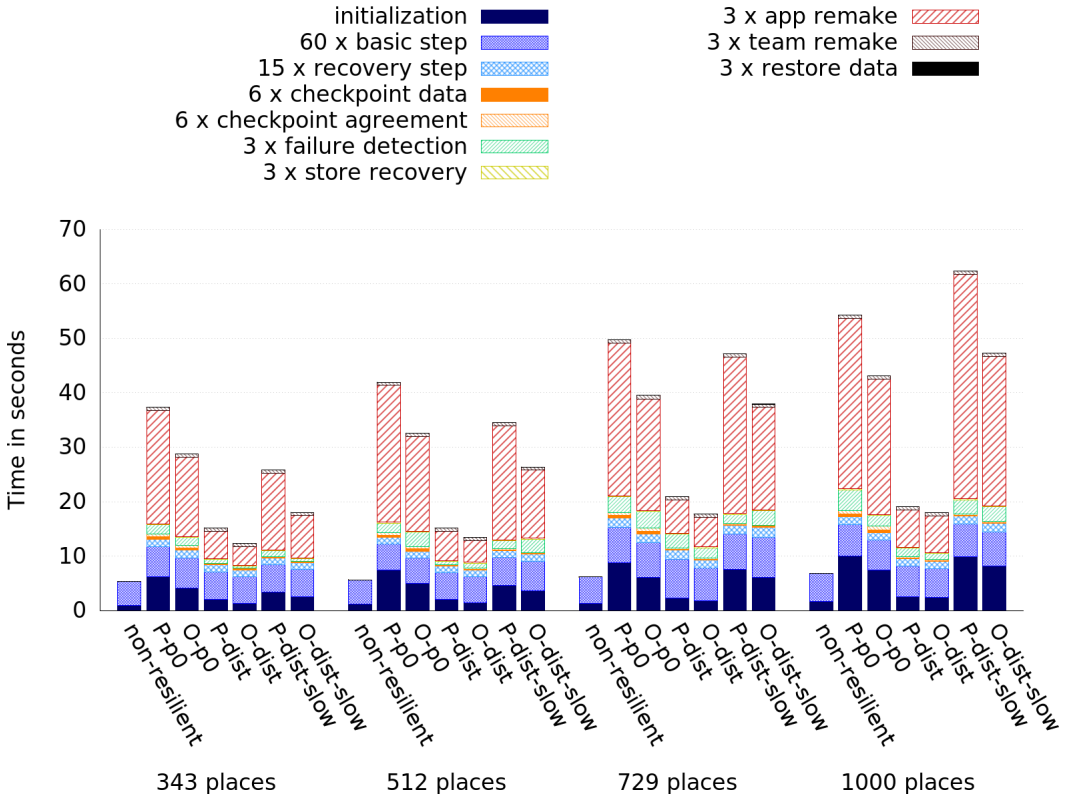

Figure 5.17: Resilient LULESH weak scaling with number of places (1 core per place).

Table 5.14: LULESH performance.

		343 places	**1000 places**
Non-res.	initialization	973 ms	1720 ms
	step	73 ms	85 ms
	total	5333 ms	6820 ms
Res. (O-dist)	initialization	1355 ms	2410 ms
	step	82 ms	89 ms
	checkpoint data (1 ckpt)	24 ms	24 ms
	checkpoint agree (1 ckpt)	31 ms	35 ms
	failure detection (1 failure)	123 ms	364 ms
	store recovery (1 failure)	48 ms	60 ms
	application remake (1 failure)	1146 ms	2243 ms
	team remake (1 failure)	198 ms	194 ms
	restore data (1 failure)	3 ms	3 ms
	total (6 ckpts, 3 failures)	12362 ms	18005 ms

Table 5.15: Changed lines of code for adding resilience using the iterative application framework.

	Total LOC	Changed LOC	Changed %
Linear regression	414	86	21%
Logistic regression	585	110	19%
PageRank	383	65	17%
LULESH	4100	200	5%

LULESH with DMTCP

At the early stage of developing the iterative application framework, we conducted a preliminary experiment to compare the performance of application-level checkpointing versus transparent checkpointing with the DMTCP [Ansel et al., 2009] tool. The DMTCP tool does not require any code changes. It checkpoints all the application data on disk even if part of this data is not needed for recovery or is computable from other values. On the other hand, our framework checkpoints in memory and relies on users to specify the values to be checkpointed. We measured the time to checkpoint LULESH with a problem size of 30^3 elements per place using 216 places on *NECTAR* virtual machines. The size of the user-defined checkpointing state is about 2.6 MB per place. The non-resilient X10 mode was used with the DMTCP runs and the resilient P-p0 mode, the only resilient mode available at that time, was used with our resilient framework runs. DMTCP took 6.67 seconds to checkpoint the entire application state, while our framework took only 0.35 seconds (19X faster than DMTCP). With DMTCP the application stops when it faces a place failure and should be restarted for recovery (ideally on the same nodes to avoid moving the checkpointing files to new locations). This limitation is avoided by our resilient framework which does not kill the application upon a failure and encapsulates the checkpoints within the application's memory.

X10 was compiled from revision `41ab27e` of the `br07_dmtcp` branch of the language repository `https://github.com/shamouda/x10.git`. LULESH was compiled from revision `8439557` of the `br07_dmtcp` branch of the applications repository `https://github.com/shamouda/x10-applications`. The X10 places used 1 worker thread by setting `X10_NTHREADS`=1 and did not use any immediate threads. The MPI threading level was configured as `MPI_THREAD_MULTIPLE`. MPICH was used as the transport layer of X10 due to technical difficulties integrating MPI-ULFM and DMTCP at that time. Because MPICH is not resilient, we could not simulate a failure recovery scenario in this experiment.

5.4.4 Summary of Performance Results

The experiments of the transactional store showed the importance of choosing the CC mechanism based on the workload characteristics. While a read-intensive application is expected to benefit more from a pessimistic concurrency control mechanism, such

as RL_EA_UL, a write-intensive application is expected to benefit more from an optimistic concurrency control mechanism, such as RV_LA_WB. Applications pay considerable performance cost for using resilient transactions; this cost increases as the transaction size and the number of write operations increase. However, this result is not surprising given the need for data replication and consensus for ensuring data availability and consistency in the presence of failures. The use of small transactions less frequently in the application is expected to result in minimal resilience overhead.

The experiments of the iterative applications demonstrated the successful role of MPI-ULFM and the optimistic termination detection protocol in reducing the resilience overhead of RX10 applications. Using native `Team` collectives does not add any measurable resilience overhead to the applications. The impact of the finish protocol on the overall performance varies between applications. The GML applications which depend only on the fan-out finish pattern do not gain a significant advantage from using the optimistic finish protocol or the distributed finish store. On the other hand, LULESH improves significantly with the optimistic finish protocol due to the communication intensity of its initialization and recovery kernels. It also scales significantly better with a distributed finish store due to creating large number of concurrent finish constructs. As expected, applications with larger checkpoint state take longer time to save a checkpoint, to agree on the checkpoint status, and to recover the resilient store.

Productivity-wise, we showed how the transaction interface has enabled us to provide atomicity and resilience to the SSCA2-k4 application with less than half the lines of code used in a non-resilient lock-based implementation. For the iterative applications, Table 5.15 shows the productivity advantage of using the resilient iterative application framework for adding resilinece to existing X10 applications with limited code changes.

5.5 Summary

This chapter presented multiple contributions that aim at reducing the burden of handling data resilience in APGAS applications.

Adding failure-awareness to the async-finish model, although necessary for recovering the control flow, is not sufficient for recovering all applications. Stateful applications often require a data resilience mechanism that preserves the application's state in spite of failures. Because handling data resilience at the application-level is a tedious and error prone task, we extended the RX10 model with resilient data stores that can protect critical application data. We described the design of two in-memory data stores that enable X10 applications to persist critical application data with minimal programming effort.

We addressed the challenge of handling atomic operations on a distributed store efficiently and correctly. This chapter described our proposed transactional **`finish`** construct, which transparently handles termination detection and transaction commitment. The availability of a transaction interface not only improves the programmer's productivity, but can also result in more trusted implementations compared to man-

ually tuned lock-based alternatives. The CC mechanism used by the transaction implementation is an important factor in the application's performance. The chapter presented a comparative performance evaluation using two CC mechanisms and two types of workloads executing in non-resilient and resilient modes. To the best of our knowledge, this is the first evaluation of resilient transactions in any PGAS and APGAS languages.

Towards the goal of supporting data resilience in RX10, we put multi-resolution resilience into practice. We showed how the resilient async-finish model, with failure awareness and composability, enables the nesting of resilient components for building high-level resilient frameworks. Two frameworks were presented; an iterative framework for bulk-synchronous applications and a parallel workers framework suitable for embarrassingly parallel applications. The performed application studies confirm the productivity advantage of these frameworks. The chapter concluded with a thorough performance analysis for these applications that evaluated the various techniques developed in this thesis for improving the scalability and efficiency of RX10 applications. In particular, it highlights the resulting performance gains from using MPI-ULFM, the optimistic finish protocol, and the distributed finish implementations.

Chapter 6

Conclusion

This thesis addressed the challenges of designing high-level parallel programming models that can exploit the massive parallelism available in modern supercomputers, while supporting programmer productivity and resilience to process failures. It proposed multi-resolution resilience as an approach for reaching a balance between the productivity of transparent system-level resilience and the potential performance scalability of user-level resilience. Multi-resolution resilience enables the construction of productive resilient frameworks at different levels of abstraction based on efficient composable resilient constructs in high-level parallel programming languages.

A common property in high-level parallel programming models is support for nested task-parallelism via composable task-parallel constructs. Such task constructs can provide a flexible base for supporting multi-resolution resilience. However, orchestrating the control flow of nested task graphs in the presence of failure while guaranteeing correctness and scalability is challenging. This thesis addressed this challenge in the context of Resilient X10 (RX10) — a resilient APGAS programming language. RX10 provides user-level resilience support by extending the async-finish model with failure-awareness semantics that facilitate reasoning about the application's control flow under failures. Our experience using RX10 shows that extending async-finish with structured failure reporting via exceptions enables adding fault tolerance to existing codes in a modular and understandable way, and also facilitates the hierarchical composition of fault tolerance strategies.

Our work identified and addressed performance and productivity limitations in RX10 that hindered its support for large-scale HPC applications. First, the resilient async-finish model imposes high failure-free resilience overhead. Second, RX10 sacrificed portability and scalability advantages available in emerging fault-tolerant communication libraries, such as MPI-ULFM, which provide optimized implementations of common communication patterns in HPC applications. Third, RX10 did not provide productive mechanisms for protecting the availability and consistency of the application data in the presence of failures, which limits the productivity of the RX10 model.

In the following, we summarize how we addressed the above limitations by answering the thesis research questions.

6.1 Answering the Research Questions

Q1: How to improve the resilience efficiency of the async-finish task model?

We identified two main reasons for the high resilience overhead of the async-finish model in RX10. The first reason is due to using a pessimistic termination detection (TD) protocol for the `finish` construct that favors the simplicity of failure recovery over the performance of failure-free execution. This pessimistic protocol uses two additional messages per task, compared to non-resilient finish, to capture every transition in the task state to avoid any uncertainty about the control flow structure when a failure occurs. The second reason is due to using a non-scalable resilient store for maintaining critical TD metadata. As an example, executing an all-to-all computation over 1024 places can result in a resilience overhead of about 5000% with the above limitations.

Our work demonstrated that the efficiency of the resilient async-finish model can be improved by adopting message-optimal TD protocols that favor the performance of failure-free execution over the performance and simplicity of failure recovery. By avoiding the communication of certain task and finish events, our proposed 'optimistic finish' 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. Compared to non-resilient finish, the optimistic protocol uses one additional message per task. By switching from the pessimistic protocol to the optimistic protocol, an all-to-all computation over 1024 places can achieve up to 50% performance improvement.

Our work demonstrated that the efficiency of the resilient async-finish model can also be improved by using a scalable resilient store for maintaining critical TD metadata (the resilient finish store). This optimization is mainly useful for computations that use large numbers of concurrent `finish` scopes. For example, a tree fan-out computation over 1024 places can achieve up to 88% performance improvement by switching from a centralized implementation of the optimistic finish protocol to a distributed one. However, our performance evaluation showed that a distributed resilient finish implementation is not always the most efficient option for certain applications. Computations that use one or a few `finish` constructs, such as flat fan-out and all-to-all, execute more efficiently using a centralized store in which replicating the TD signals is not required. For example, the all-to-all pattern, which uses a single `finish` to track a large number of tasks, achieves up to 39% performance improvement by switching from a distributed to a centralized optimistic finish implementation.

It is worth noting that the survivability of the optimistic protocol and the pessimistic protocol is limited to the survivability of the resilient finish store. With the centralized store, the runtime can survive all failures except the failure of place zero, which is used for saving all TD metadata. With the distributed store, the runtime can survive all failures except the simultaneous failure of a place and its backup.

The performance of different computation patterns, as summarized in Table 4.3, can guide the choice of the most suitable protocol-implementation combination for RX10 applications. For the shock hydrodynamics application LULESH, the best performance was achieved using the optimistic distributed finish implementation. By switching from the centralized pessimistic finish implementation (P-p0) to our proposed distributed optimistic finish implementation (O-dist), the resilience overhead of a single step of LULESH reduced from 13% to only 4% (see Section 5.4.3.5).

Q2: How to exploit the fault tolerance capabilities of MPI-ULFM to improve the scalability of resilient APGAS languages?

Our work found the following benefits in MPI-ULFM that make it a suitable base for resilient APGAS languages:

- The asynchronous execution model of APGAS languages makes global synchronization for failure detection or runtime recovery inefficient or even infeasible for some languages. While global failure propagation and global failure recovery are supported, MPI-ULFM does not impose such synchronous failure handling mechanisms on its clients by default. Asynchronous programming languages can achieve global failure detection while avoiding global synchronization by periodically checking for incoming messages from any other process using `MPI_Iprobe` and `MPI_ANY_SOURCE`.
- Many APGAS applications follow the SPMD execution model and express their algorithms using collective operations. As an extension of MPI-3, MPI-ULFM provides efficient fault-tolerant collective functions, which can be used by bulk-synchronous APGAS applications for achieving better performance.
- MPI-ULFM provides an efficient fault-tolerant agreement interface that applications can leverage for implementing different fault tolerance mechanisms.
- Finally, our experimental evaluation shows that MPI-ULFM does not impose a significant resilience overhead on applications in failure-free executions.

The above capabilities facilitated the integration between RX10 and MPI-ULFM as described in Chapter 3. A direct advantage of using MPI-ULFM as a communication runtime for RX10 is enabling applications to utilize the superior networking and compute capabilities of large-scale supercomputers in which MPI is widely supported. More importantly, exploiting the optimized fault-tolerant collective operations provided by MPI-ULFM enabled us to eliminate most of the resilience overhead of X10's bulk-synchronous applications, which would otherwise use emulated collectives that are subject to the resilience overhead of the async-finish model. For example, the Linear Regression and Logistic Regression benchmarks, which rely entirely on collective operations for step execution, achieve almost the same step execution time in non-resilient and resilient modes (see Section 5.4.3.2 and Section 5.4.3.3). By using the fault-tolerant agreement capability of MPI-ULFM, coordinated checkpointing has been supported for RX10 applications with minimal resilience overhead.

At the time of writing, implementation of fault-tolerant one-sided communication functions in MPI-ULFM has not been completed yet. Because X10 relies on two-sided communication functions, evaluating MPI-ULFM's support for one-sided communication has been out of the scope of this thesis.

Q3: How to improve the productivity of resilient APGAS languages that support user-level fault tolerance?

This thesis highlighted two main sources of complexities that face developers of fault-tolerant applications: protecting the availability and consistency of the application data and handling application recovery using fewer resources (i.e. shrinking recovery). We demonstrated that these complexities can be greatly simplified by providing resilient data store abstractions that support the following features: strong locality, in-memory replication, non-shrinking recovery, and support for distributed transactions.

This thesis demonstrated that the composable task-parallel constructs provided by APGAS languages, if extended with failure awareness, enable the composition of high-level resilient frameworks that hide most of the fault tolerance complexities from the programmer. It also showed that the resilient TD protocols used by the async-finish model can easily be extended to coordinate distributed transaction commitment.

To the best of our knowledge, support for resilient distributed transactions has not been attempted for PGAS or APGAS languages prior to our work. This thesis filled this gap by proposing a transactional finish construct as productive mechanism for handling resilient atomic updates on global data. The transactional finish construct enables the expression of dynamic transactions of arbitrary sizes and is flexible to different CC mechanisms. Our performance evaluation showed that the overhead of resilient transactions depends on the number of transaction participants and the number of conflicts that impact a transaction. We implemented a variety of CC mechanisms to enable programmers to control the conflict rate of their applications. Our results showed that write-intensive transactions can achieve higher throughput by relying on an optimistic concurrency control mechanism that delays acquiring data locks to commitment time (see Figure 5.12). In contrast, read-intensive transactions can achieve higher throughput by eagerly acquiring data locks during transaction processing (see Figure 5.11).

6.2 Future Work

6.2.1 Fault-Tolerant One-Sided Communication

An untouched problem in this thesis is exploiting recent advances in fault-tolerant one-sided communication libraries for building high-level resilient languages. Unlike X10, which is implemented entirely using two-sided communication operations, most PGAS languages provide productive data access operations based on implicit transfer of remote data using one-sided get/put operations. Unfortunately, GASNet, the most commonly used communication library for PGAS programming models, still lacks

fault tolerance support. The availability of a resilient GASNet implementation could enable many PGAS languages to add support for resilience. Meanwhile, research in the GASPI [Simmendinger et al., 2015] library is promising. GASPI provides fault-tolerant one-sided communication functions that could be used for building resilient PGAS models. One research direction that we aim to investigate is the design of user-level fault tolerance in PGAS languages, such as Chapel and UPC, based on GASPI, and comparing the performance of these models to the performance of RX10.

6.2.2 Hardware Support for Distributed Transactions

Improving the performance of the transactional finish construct is an interesting area for future investigation. Emerging approaches for supporting distributed transactions in PGAS-based distributed systems rely on exploiting hardware capabilities, such as hardware transactional memory (HTM) and RDMA, for achieving higher processing throughput [Dragojević et al., 2015; Chen et al., 2016]. While we currently implement transaction management at the user level, the transactional finish construct could transparently leverage such hardware techniques for achieving higher throughput.

6.2.3 Beyond X10

Although the thesis has focused on X10, its contributions can be applied to other programming languages. The optimistic finish protocol can be used in dynamic task-based systems that require termination detection. For example, Chapel's `sync` statement and `begin` construct can express async-finish-like task graphs. We expect applying the optimistic finish protocol to be straightforward in this context. The transactional finish construct, the design of the resilient store, and the resilient application frameworks are also generic and can be easily implemented in other languages. We provided a fully functional implementation of RX10 over MPI-ULFM, which serves as a basis for future performance comparisons as more languages start to support resilience. We expect future development of MPI-ULFM to include implementations of fault tolerant one-sided communication functions. This will provide an opportunity for evaluating the suitability of the fault tolerance model of MPI-ULFM to more PGAS languages.

Appendices

Appendix A

Evaluation Platforms

Evaluation was performed on two different parallel machines:

- *Raijin*: a supercomputer hosted at the Australian National Computing Infrastructure (NCI), at the Australian National University. Each compute node in Raijin has two 8-core Intel Xeon (Sandy Bridge 2.6 GHz) sockets, and uses an Infiniband FDR network. Each core has two hardware threads; however, Raijin disables hyper-threading for the PBS jobs by default to keep one hardware thread available for the operating system tasks on each core.

- *NECTAR*: a cloud service provider that assembles compute nodes from multiple Australian instituations, including NCI. An NCI NECTAR node has an Intel Xeon CPU E5-2670 @ 2.60GHz processor and connects to other nodes through an Infiniband 56 GbE (Mellanox) network. We have access to 30 virtual machines of size 'm2.xlarge', all hosted at the NCI zone. An m2.xlarge virtual machine has 45 GiB of memory and 12 virtual cores, providing a total of 360 virtual cores for our experiments. Our virtual machines use NECTAR Ubuntu 14.04 (Trusty) amd64 operating system.

In our experiments on *Raijin*: we allocated 10 GiB of memory per node (unless otherwise stated), and used the default hyper-threading configuration. Our MPI jobs used the OpenIB byte transfer layer , used static core binding, and mapped the processes over the nodes in a round-robin manner using the following `mpirun` parameters: `-bind-to core -map-by node -mca btl openib,vader,self`. Both OpenMPI 3.0.0, and MPI-ULFM2 on Raijin use the default eager limit `btl_openib_eager_limit` as 12288 bytes.

In our experiments on *NECTAR*: MPI-ULFM2 used the TCP byte transfer layer, used static core binding, and mapped the processes over the nodes in a round-robin manner using the following `mpirun` parameters: `-bind-to core -map-by node`. We modified the eager limit `btl_tcp_eager_limit` to be 12288 bytes, to match Raijin's configuration.

For all reported results, we used the Native (C++ backend) version of X10 compiled using gcc 4.4.7. Unless otherwise specified, each X10 place was configured to

use only one worker thread by setting the environment variable `X10_NTHREADS`=1. Experiments that used MPI as a transport layer used an additional 'immediate thread' per place to handles low-level non-blocking active messages invoked by the finish protocols by setting `X10_NUM_IMMEDIATE_THREADS`=1. The MPI threading level was configured as `MPI_THREAD_SERIALIZED` by setting `X10RT_MPI_THREAD_SERIALIZED`=1.

MPI-ULFM2 was installed from revision `e87f595` of the master repository at `https://bitbucket.org/icldistcomp/ulfm2.git`

Appendix B

TLA+ Specification of the Optimistic Finish Protocol

```
1 ---------- MODULE Optimistic ----------
```

This specification models the 'optimistic *finish*' protocol used for detecting the termination of async-finish task graphs. We model the graph as connected nodes of tasks. Finish objects do not represent separate nodes in the task graph, but implicit objects attached to tasks.

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 c children. The variables $LEVEL$, NUM_PLACES and $WIDTH$ can be used to configure the graph. The model also permits simulating 0 or more failures by configuring the MAX_KILL variable.

For the model checker to generate all possible execution scenarios, it can run out of memory, especially when activating failure recovery actions. We introduced the variables $KILL\_FROM$ and $KILL\_TO$ to control the range of steps at which failures can occur, so that we can cut the verification process into multiple phases. For example, we used 4 phases to simulate all possible execution scenarios for a 3-level

3-place task tree with width 2, that takes around 50 steps in total:

Phase 1: kills a place between steps 0 and 20.

Phase 2: kills a place between steps 20 and 30.

Phase 3: kills a place between steps 30 and 50.

Phase 4: kills a place between steps 50 and 100.

See the run figures at: https://*github.com*/shamouda/x10-formal-spec/tree/master/async-finish-optimistic/*run_figures*

```
31 EXTENDS Integers

33 CONSTANTS
34  LEVEL,      task tree levels
35  WIDTH,      task tree branching factor
36  NUM_PLACES, number of places
37  MAX_KILL,   maximum number of failures to simulate
38  KILL_FROM, KILL_TO  the range of steps to simulate a failure at

40 VARIABLES
```

```
41  exec_state,  execution state
42  tasks,  set of tasks
43  f_set,  finish objects
44  lf_set,  local finish objects
45  rf_set,  resilient finish objects
46  msgs,  msgs
47  nxt_finish_id,  sequence of finish ids
48  nxt_task_id,  sequence of task ids
49  nxt_remote_place,  next place to communicate with
50  killed,  set of killed places
51  killed_cnt,  size of the killed set
52  rec_child,  pending recovery actions: ghosts queries
53  rec_to,  pending recovey actions: ignoring tasks to dead places
54  rec_from,  pending recovey actions: counting messages from dead places
55  rec_from_waiting,  pernding recovery actions: receiving counts of messages from dead places
56  lost_tasks,  debug variable: set of lost tasks due to a failure
57  lost_f_set,  debug variable: set of lost finishes
58  lost_lf_set,  debug variable: set of lost local finishes
59  step  the exectution step of the model

61  Vars ≜ ⟨exec_state, tasks, f_set, lf_set, rf_set, msgs,
62          nxt_finish_id, nxt_task_id, nxt_remote_place,
63          killed, killed_cnt,
64          lost_tasks, lost_f_set, lost_lf_set,
65          rec_child, rec_to, rec_from, rec_from_waiting, step⟩

67  ----------------------------------------------------------------
68  C ≜ INSTANCE OptimisticCommons
69  ----------------------------------------------------------------
70  TypeOK ≜
      Variables type constrains
74    ∧ exec_state ∈ {"running", "success"}
75    ∧ tasks ⊆ C!Task
76    ∧ f_set ⊆ C!Finish
77    ∧ lf_set ⊆ C!LFinish
78    ∧ rf_set ⊆ C!RFinish
79    ∧ nxt_finish_id ∈ C!FinishID
80    ∧ nxt_task_id ∈ C!TaskID
81    ∧ nxt_remote_place ∈ C!Place1D
82    ∧ killed ⊆ C!PlaceID
83    ∧ killed_cnt ∈ 0 .. (NUM_PLACES − 1)
84    ∧ rec_child ⊆ C!GetChildrenTask
85    ∧ rec_to ⊆ C!ConvTask
86    ∧ rec_from ⊆ C!ConvTask
```

```
87    ∧ rec_from_waiting ⊆ C!ConvTask
88    ∧ step ∈ Nat

90  ├──────────────────────────────────────────────────────────────┤
91  MustTerminate ≜
      Temporal property: the program must eventually terminate successfully
95    ◇(exec_state = "success")

97  ├──────────────────────────────────────────────────────────────┤
98  Init ≜
      Initialize variables
102   ∧ exec_state = "running"
103   ∧ tasks = {C!RootTask, C!RootFinishTask}
104   ∧ f_set = {C!RootFinish}
105   ∧ lf_set = {}
106   ∧ rf_set = {}
107   ∧ msgs = {}
108   ∧ nxt_finish_id = 2
109   ∧ nxt_task_id = 2
110   ∧ nxt_remote_place = [i ∈ C!PlaceID ↦ i]
111   ∧ killed = {}
112   ∧ killed_cnt = 0
113   ∧ lost_tasks = {}
114   ∧ lost_f_set = {}
115   ∧ lost_lf_set = {}
116   ∧ rec_child = {}
117   ∧ rec_to = {}
118   ∧ rec_from = {}
119   ∧ rec_from_waiting = {}
120   ∧ step = 0

122 ├──────────────────────────────────────────────────────────────┤
    Utility actions: creating instances of task, finish, resilient finish and local finish
127 NewFinish(task) ≜
128 [id ↦ nxt_finish_id,
129  pred_id ↦ task.id,
130  home ↦ task.dst,
131  origin ↦ task.src,
132  parent_finish_id ↦ task.finish_id,
133  status ↦ "active",
134  lc ↦ 1  the main finish task
135 ]

137 NewResilientFinish(finish) ≜
138 [id ↦ finish.id,
```

```
 home ↦ finish.home,
 origin ↦ finish.origin,
 parent_finish_id ↦ finish.parent_finish_id,
 transOrLive ↦ C!Place2DInitResilientFinish(finish.home),
 sent ↦ C!Place2DInitResilientFinish(finish.home),
 gc ↦ 1,
 ghost_children ↦ {},
 isAdopted ↦ FALSE]

NewLocalFinish(fid, dst) ≜
[id ↦ fid,
 home ↦ dst,
 lc ↦ 0,
 reported ↦ C!Place1DZeros,
 received ↦ C!Place1DZeros,
 deny ↦ C!Place1DZeros]

NewTask(pred, fid, s, d, t, l, st, fin_type) ≜
[id ↦ nxt_task_id,
 pred_id ↦ pred,
 src ↦ s,
 dst ↦ d,
 finish_id ↦ fid,
 level ↦ l,
 last_branch ↦ 0,
 status ↦ st,
 type ↦ t,
 finish_type ↦ fin_type]
```

Finish Actions

```
Task_CreatingFinish ≜
  ∧ exec_state = "running"
  ∧ LET task ≜ C!FindRunningTask(LEVEL − 1)
        task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
                       ELSE [task EXCEPT !.last_branch = task.last_branch + 1,
                                         !.status = "blocked"]
        finish ≜ IF task ≠ C!NOT_TASK
                 THEN NewFinish(task)
                 ELSE C!NOT_FINISH
        finish_task ≜ IF task ≠ C!NOT_TASK
                      THEN NewTask(task.id, finish.id, task.dst, task.dst,
                                   "finishMainTask", task.level + 1, "running", "global")
                      ELSE C!NOT_TASK
    IN  ∧ task ≠ C!NOT_TASK
```

```
186          ∧ nxt_finish_id' = nxt_finish_id + 1
187          ∧ nxt_task_id' = nxt_task_id + 1
188          ∧ f_set' = f_set ∪ {finish}
189          ∧ tasks' = (tasks \ {task}) ∪ {task_updated, finish_task}
190          ∧ step' = step + 1
191    ∧ UNCHANGED ⟨exec_state, lf_set, rf_set, msgs,
192                 nxt_remote_place,
193                 killed, killed_cnt,
194                 lost_tasks, lost_f_set, lost_lf_set,
195                 rec_child, rec_to, rec_from, rec_from_waiting⟩

197  Finish_CreatingRemoteTask ≜
198    ∧ exec_state = "running"
199    ∧ LET task ≜ C!FindRunningTaskWithFinishType(LEVEL − 1, "global")
200          task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
201                         ELSE [task EXCEPT !.last_branch = task.last_branch + 1,
202                                           !.status = "blocked"]
203          finish ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
204                   ELSE C!FindFinishById(task.finish_id)
205          new_finish_status ≜ IF C!IsPublished(task.finish_id)
206                              THEN finish.status
207                              ELSE "waitingForPublish"
208          finish_updated ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
209                           ELSE [finish EXCEPT !.status = new_finish_status]
210          src ≜ task.dst
211          dst ≜ C!NextRemotePlace(src)
212          new_task_status ≜ IF C!IsPublished(task.finish_id)
213                            THEN "waitingForTransit"
214                            ELSE "waitingForPublish"
215          new_task ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
216                     ELSE NewTask(task.id, task.finish_id, src, dst,
217                                  "normal", task.level + 1, new_task_status, "local")
218          msg_transit ≜ [from ↦ "src", to ↦ "rf", tag ↦ "transit",
219                         src ↦ new_task.src, dst ↦ new_task.dst,
220                         finish_id ↦ new_task.finish_id,
221                         task_id ↦ new_task.id]
222          msg_publish ≜ [from ↦ "f", to ↦ "rf", tag ↦ "publish",
223                         src ↦ finish.home,
224                         finish_id ↦ finish.id]
225      IN ∧ task ≠ C!NOT_TASK
226         ∧ finish.status = "active"
227         ∧ nxt_task_id' = nxt_task_id + 1
228         ∧ tasks' = (tasks \ {task}) ∪ {task_updated, new_task}
229         ∧ f_set' = (f_set \ {finish}) ∪ {finish_updated}
```

```
230         ∧ C!ShiftNextRemotePlace(src)
231         ∧ IF C!IsPublished(task.finish_id)
232           THEN C!SendMsg(msg_transit)
233           ELSE C!SendMsg(msg_publish)
234         ∧ step' = step + 1
235   ∧ UNCHANGED ⟨exec_state, lf_set, rf_set,
236                nxt_finish_id,
237                killed, killed_cnt,
238                lost_tasks, lost_f_set, lost_lf_set,
239                rec_child, rec_to, rec_from, rec_from_waiting⟩

241 Finish_ReceivingPublishDoneSignal ≜
242   ∧ exec_state = "running"
243   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("f", "publishDone")
244         finish ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
245                  ELSE C!FindFinishById(msg.finish_id)
246         finish_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
247                          ELSE [finish EXCEPT !.status = "active"]
248         pending_task ≜ C!FindPendingRemoteTask(finish.id, "waitingForPublish")
249         pending_task_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
250                                ELSE [pending_task EXCEPT !.status = "waitingForTransit"]
251         msg_transit ≜ [from ↦ "src", to ↦ "rf", tag ↦ "transit",
252                        src ↦ pending_task.src, dst ↦ pending_task.dst,
253                        finish_id ↦ pending_task.finish_id,
254                        task_id ↦ pending_task.id]
255     IN  ∧ msg ≠ C!NOT_MESSAGE
256         ∧ C!ReplaceMsg(msg, msg_transit)
257         ∧ f_set' = (f_set \ {finish}) ∪ {finish_updated}
258         ∧ tasks' = (tasks \ {pending_task}) ∪ {pending_task_updated}
259         ∧ step' = step + 1
260   ∧ UNCHANGED ⟨exec_state, lf_set, rf_set,
261                nxt_finish_id, nxt_task_id, nxt_remote_place,
262                killed, killed_cnt,
263                lost_tasks, lost_f_set, lost_lf_set,
264                rec_child, rec_to, rec_from, rec_from_waiting⟩

266 Finish_TerminatingTask ≜
267   ∧ exec_state = "running"
268   ∧ LET task ≜ C!FindTaskToTerminate("global")
269         finish ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
270                  ELSE C!FindFinishById(task.finish_id)
271         task_updated ≜ IF task ≠ C!NOT_TASK
272                        THEN [task EXCEPT !.status = "terminated"]
273                        ELSE C!NOT_TASK
274         finish_updated ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
```

```
                    ELSE IF finish.lc = 1 ∧ C!IsPublished(finish.id)
                    THEN [finish EXCEPT !.lc = finish.lc − 1,
                                        !.status = "waitingForRelease"]
                    ELSE IF finish.lc = 1 ∧ ¬C!IsPublished(finish.id)
                    THEN [finish EXCEPT !.lc = finish.lc − 1,
                                        !.status = "released"]
                    ELSE [finish EXCEPT !.lc = finish.lc − 1]
  IN ∧ task ≠ C!NOT_TASK
     ∧ finish ≠ C!NOT_FINISH
     ∧ f_set' = (f_set \ {finish}) ∪ {finish_updated}
     ∧ IF finish_updated.status = "waitingForRelease"
       THEN msgs' = msgs ∪
                    {[from ↦ "f", to ↦ "rf", tag ↦ "terminateTask",
                      src ↦ finish.home,
                      finish_id ↦ finish.id,
                      task_id ↦ task.id,
                      term_tasks_by_src ↦ C!Place1DTerminateTask(finish.home, 1),
                      term_tasks_dst ↦ finish.home]}
       ELSE msgs' = msgs
     ∧ IF finish_updated.status = "released"
       THEN LET task_blocked ≜ C!FindBlockedTask(finish.pred_id)
                task_unblocked ≜ [task_blocked EXCEPT !.status = "running"]
            IN tasks' = (tasks \ {task, task_blocked})
                        ∪ {task_updated, task_unblocked}
       ELSE tasks' = (tasks \ {task}) ∪ {task_updated}
     ∧ step' = step + 1
  ∧ UNCHANGED ⟨exec_state, lf_set, rf_set,
               nxt_finish_id, nxt_task_id, nxt_remote_place,
               killed, killed_cnt,
               lost_tasks, lost_f_set, lost_lf_set,
               rec_child, rec_to, rec_from, rec_from_waiting⟩

Finish_ReceivingReleaseSignal ≜
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("f", "release")
        finish ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
                 ELSE C!FindFinishToRelease(msg.finish_id)
        finish_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
                         ELSE [finish EXCEPT !.status = "released"]
        task_blocked ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
                       ELSE C!FindBlockedTask(finish.pred_id)
        task_unblocked ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
                         ELSE [task_blocked EXCEPT !.status = "running"]
    IN ∧ msg ≠ C!NOT_MESSAGE
```

```
319         ∧ C!RecvMsg(msg)
320         ∧ f_set' = (f_set \ {finish}) ∪ {finish_updated}
321         ∧ tasks' = (tasks \ {task_blocked}) ∪ {task_unblocked}
322         ∧ step' = step + 1
323   ∧ UNCHANGED ⟨exec_state, lf_set, rf_set,
324                nxt_finish_id, nxt_task_id, nxt_remote_place,
325                killed, killed_cnt,
326                lost_tasks, lost_f_set, lost_lf_set,
327                rec_child, rec_to, rec_from, rec_from_waiting⟩

329 ----------------------------------------------------------------
    Actions applicable to Finish and Local Finish
333 DroppingTask ≜
334   ∧ exec_state = "running"
335   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("src", "transitNotDone")
336         task ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
337                  ELSE C!FindTaskById(msg.task_id)
338         task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
339                        ELSE [task EXCEPT !.status = "dropped"]
340         blocked_task ≜ C!FindTaskById(task.pred_id)
341         blocked_task_updated ≜ [blocked_task EXCEPT !.status = "running"]
342     IN  ∧ msg ≠ C!NOT_MESSAGE
343         ∧ task.status = "waitingForTransit"
344         ∧ blocked_task.status = "blocked"
345         ∧ tasks' = (tasks \ {task, blocked_task}) ∪
346                    {task_updated, blocked_task_updated}
347         ∧ C!RecvMsg(msg)
348         ∧ step' = step + 1
349   ∧ UNCHANGED ⟨exec_state, f_set, lf_set, rf_set,
350                nxt_finish_id, nxt_task_id, nxt_remote_place,
351                killed, killed_cnt,
352                lost_tasks, lost_f_set, lost_lf_set,
353                rec_child, rec_to, rec_from, rec_from_waiting⟩

355 SendingTask ≜
356   ∧ exec_state = "running"
357   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("src", "transitDone")
358         task ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
359                  ELSE C!FindTaskById(msg.task_id)
360         task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
361                        ELSE [task EXCEPT !.status = "sent"]
362         blocked_task ≜ C!FindTaskById(task.pred_id)
363         blocked_task_updated ≜ [blocked_task EXCEPT !.status = "running"]
364     IN  ∧ msg ≠ C!NOT_MESSAGE
```

```
365         ∧ task.status = "waitingForTransit"
366         ∧ blocked_task.status = "blocked"
367         ∧ tasks' = (tasks \ {task, blocked_task}) ∪
368                        {task_updated, blocked_task_updated}
369         ∧ C!ReplaceMsg(msg, [from ↦ "src", to ↦ "dst", tag ↦ "task",
370                              src ↦ task.src, dst ↦ task.dst,
371                              finish_id ↦ task.finish_id,
372                              task_id ↦ task.id])
373         ∧ step' = step + 1
374   ∧ UNCHANGED ⟨exec_state, f_set, lf_set, rf_set,
375                nxt_finish_id, nxt_task_id, nxt_remote_place,
376                killed, killed_cnt,
377                lost_tasks, lost_f_set, lost_lf_set,
378                rec_child, rec_to, rec_from, rec_from_waiting⟩

380 ReceivingTask ≜
381   ∧ exec_state = "running"
382   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("dst", "task")
383         src ≜ msg.src
384         dst ≜ msg.dst
385         finish_id ≜ msg.finish_id
386         lfinish ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
387                   ELSE IF C!LocalFinishExists(dst, finish_id)
388                   THEN C!FindLocalFinish(dst, finish_id)
389                   ELSE NewLocalFinish(finish_id, dst)
390         lfinish_updated ≜ [lfinish EXCEPT
391                            !.received[src] = lfinish.received[src] + 1,
392                            !.lc = lfinish.lc + 1]
393         task ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_TASK
394                ELSE C!FindTaskById(msg.task_id)
395         task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
396                        ELSE [task EXCEPT !.status = "running"]
397     IN  ∧ msg ≠ C!NOT_MESSAGE
398         ∧ C!RecvMsg(msg)
399         ∧ IF lfinish.deny[src] = 1
400           THEN ∧ lf_set' = lf_set
401                ∧ tasks' = tasks
402           ELSE ∧ lf_set' = (lf_set \ {lfinish}) ∪ {lfinish_updated}
403                ∧ tasks' = (tasks \ {task}) ∪ {task_updated}
404         ∧ step' = step + 1
405   ∧ UNCHANGED ⟨exec_state, f_set, rf_set,
406                nxt_finish_id, nxt_task_id, nxt_remote_place,
407                killed, killed_cnt,
408                lost_tasks, lost_f_set, lost_lf_set,
```

```
409                 rec_child, rec_to, rec_from, rec_from_waiting⟩
411 ├──────────────────────────────────────────────────────────────┤
    Local Finish Actions
415 LocalFinish_CreatingRemoteTask ≜
416   ∧ exec_state = "running"
417   ∧ LET task ≜ C!FindRunningTaskWithFinishType(LEVEL − 1, "local")
418         task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
419                        ELSE [task EXCEPT !.last_branch = task.last_branch + 1,
420                                          !.status = "blocked"]
421         finish ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
422                  ELSE C!FindFinishById(task.finish_id)
423         src ≜ task.dst
424         dst ≜ C!NextRemotePlace(src)
425         new_task ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
426                    ELSE NewTask(task.id, task.finish_id, src, dst,
427                                 "normal", task.level + 1,
428                                 "waitingForTransit", "local")
429         msg_transit ≜ [from ↦ "src", to ↦ "rf", tag ↦ "transit",
430                        src ↦ new_task.src, dst ↦ new_task.dst,
431                        finish_id ↦ new_task.finish_id,
432                        task_id ↦ new_task.id]
433     IN  ∧ task ≠ C!NOT_TASK
434         ∧ nxt_task_id′ = nxt_task_id + 1
435         ∧ tasks′ = (tasks \ {task}) ∪ {task_updated, new_task}
436         ∧ C!ShiftNextRemotePlace(src)
437         ∧ C!SendMsg(msg_transit)
438         ∧ step′ = step + 1
439   ∧ UNCHANGED ⟨exec_state, f_set, lf_set, rf_set,
440                nxt_finish_id,
441                killed, killed_cnt,
442                lost_tasks, lost_f_set, lost_lf_set,
443                rec_child, rec_to, rec_from, rec_from_waiting⟩

445 LocalFinish_TerminatingTask ≜
446   ∧ exec_state = "running"
447   ∧ LET task ≜ C!FindTaskToTerminate("local")
448         task_updated ≜ IF task = C!NOT_TASK THEN C!NOT_TASK
449                        ELSE [task EXCEPT !.status = "terminated"]
450         here ≜ task.dst
451         finish_id ≜ task.finish_id
452         lfinish ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
453                   ELSE C!FindLocalFinish(here, finish_id)
454         lfinish_updated ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
```

```
455                         ELSE [lfinish EXCEPT !.lc = lfinish.lc − 1]
456        term_tasks ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
457                     ELSE [i ∈ C!PlaceID ↦
458                           IF i = lfinish.home THEN 0
459                            ELSE lfinish.received[i] − lfinish.reported[i] ]
460        lfinish_terminated ≜ IF task = C!NOT_TASK THEN C!NOT_FINISH
461                             ELSE [lfinish EXCEPT
462                                     !.lc = 0,
463                                     !.reported = lfinish.received]
464    IN  ∧ task ≠ C!NOT_TASK
465        ∧ lfinish ≠ C!NOT_FINISH
466        ∧ IF lfinish_updated.lc = 0
467          THEN ∧ msgs′ = msgs ∪
468                      {[from ↦ "f", to ↦ "rf", tag ↦ "terminateTask",
469                        src ↦ here,
470                        finish_id ↦ finish_id,
471                        task_id ↦ task.id,
472                        term_tasks_by_src ↦ term_tasks,
473                        term_tasks_dst ↦ here]}
474               ∧ lf_set′ = (lf_set \ {lfinish}) ∪ {lfinish_terminated}
475          ELSE ∧ msgs′ = msgs
476               ∧ lf_set′ = (lf_set \ {lfinish}) ∪ {lfinish_updated}
477        ∧ tasks′ = (tasks \ {task}) ∪ {task_updated}
478        ∧ step′ = step + 1
479  ∧ UNCHANGED ⟨exec_state, f_set, rf_set,
480               nxt_finish_id, nxt_task_id, nxt_remote_place,
481               killed, killed_cnt,
482               lost_tasks, lost_f_set, lost_lf_set,
483               rec_child, rec_to, rec_from, rec_from_waiting⟩

485 LocalFinish_MarkingDeadPlace ≜
486  ∧ exec_state = "running"
487  ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("dst", "countDropped")
488        finish_id ≜ msg.finish_id
489        here ≜ msg.dst
490        dead ≜ msg.src
491        lfinish ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
492                  ELSE IF C!LocalFinishExists(here, finish_id)
493                  THEN C!FindLocalFinish(here, finish_id)
494                  ELSE NewLocalFinish(finish_id, here)
495        lfinish_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
496                          ELSE [lfinish EXCEPT !.deny[dead] = 1]
497        resp ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_MESSAGE
498               ELSE [from ↦ "dst", to ↦ "rf", tag ↦ "countDroppedDone",
```

```
                        finish_id ↦ msg.finish_id,
                        src ↦ msg.src, dst ↦ msg.dst,
                        num_dropped ↦ msg.num_sent − lfinish.received[dead]]
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ C!ReplaceMsg(msg, resp)
        ∧ lf_set' = (lf_set \ {lfinish}) ∪ {lfinish_updated}
        ∧ step' = step + 1
  ∧ UNCHANGED ⟨exec_state, tasks, f_set, rf_set,
               nxt_finish_id, nxt_task_id, nxt_remote_place,
               killed, killed_cnt,
               lost_tasks, lost_f_set, lost_lf_set,
               rec_child, rec_to, rec_from, rec_from_waiting⟩
```

Resilient Store Actions

```
Store_ReceivingPublishSignal ≜
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("rf", "publish")
        finish_home ≜ msg.src
        finish ≜ IF msg = C!NOT_MESSAGE ∨ finish_home ∈ killed
                 THEN C!NOT_FINISH
                 ELSE C!FindFinishById(msg.finish_id)
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ IF finish_home ∉ killed
          THEN ∧ C!ReplaceMsg(msg, [from ↦ "rf", to ↦ "f",
                                    tag ↦ "publishDone",
                                    dst ↦ msg.src,
                                    finish_id ↦ msg.finish_id])
               ∧ rf_set' = rf_set ∪ {NewResilientFinish(finish)}
          ELSE ∧ C!RecvMsg(msg)
               ∧ rf_set' = rf_set
        ∧ step' = step + 1
  ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
               nxt_finish_id, nxt_task_id, nxt_remote_place,
               killed, killed_cnt,
               lost_tasks, lost_f_set, lost_lf_set,
               rec_child, rec_to, rec_from, rec_from_waiting⟩

Store_ReceivingTransitSignal ≜
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("rf", "transit")
        rf ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
             ELSE C!FindResilientFinishById(msg.finish_id)
        s ≜ msg.src
```

```
545         d ≜ msg.dst
546         rf_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
547                      ELSE [rf EXCEPT
548                              !.sent[s][d] = rf.sent[s][d] + 1,
549                              !.transOrLive[s][d] = rf.transOrLive[s][d] + 1,
550                              !.gc = rf.gc + 1]
551         msg_tag ≜ IF s ∈ killed ∨ d ∈ killed
552                   THEN "transitNotDone"
553                   ELSE "transitDone"
554     IN  ∧ msg ≠ C!NOT_MESSAGE
555         ∧ ¬C!IsRecoveringTasksToDeadPlaces(rf.id)
556         ∧ IF s ∈ killed ∨ d ∈ killed
557           THEN rf_set' = rf_set
558           ELSE rf_set' = (rf_set \ {rf}) ∪ {rf_updated}
559         ∧ C!ReplaceMsg(msg, [from ↦ "rf", to ↦ "src", tag ↦ msg_tag,
560                             dst ↦ s,
561                             finish_id ↦ msg.finish_id,
562                             task_id ↦ msg.task_id])
563         ∧ step' = step + 1
564   ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
565                nxt_finish_id, nxt_task_id, nxt_remote_place,
566                killed, killed_cnt,
567                lost_tasks, lost_f_set, lost_lf_set,
568                rec_child, rec_to, rec_from, rec_from_waiting⟩

570 Store_ReceivingTerminateTaskSignal ≜
571   ∧ exec_state = "running"
572   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("rf", "terminateTask")
573         term_tasks ≜ msg.term_tasks_by_src
574         dst ≜ msg.term_tasks_dst
575         rf ≜ IF msg = C!NOT_MESSAGE ∨ dst ∈ killed THEN C!NOT_FINISH
576              ELSE C!FindResilientFinishById(msg.finish_id)
577         trans_live_updated ≜ [i ∈ C!PlaceID ↦ [j ∈ C!PlaceID ↦
578                                IF j = dst THEN rf.transOrLive[i][j] - term_tasks[i]
579                                ELSE rf.transOrLive[i][j]
580                               ]]
581         total ≜ C!Sum(term_tasks)
582         rf_updated ≜ IF msg = C!NOT_MESSAGE ∨ dst ∈ killed THEN C!NOT_FINISH
583                      ELSE [rf EXCEPT !.transOrLive = trans_live_updated,
584                                      !.gc = rf.gc - total]
585     IN  ∧ msg ≠ C!NOT_MESSAGE
586         ∧ total ≠ -1  see C!Sum() definition
587         ∧ IF dst ∉ killed
588           THEN ∧ ¬C!IsRecoveringTasksToDeadPlaces(rf.id)
```

```
589                 ∧ C!ApplyTerminateSignal(rf, rf_updated, msg)
590           ELSE  C!RecvTerminateSignal(msg)
591         ∧ step' = step + 1
592   ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
593                nxt_finish_id, nxt_task_id, nxt_remote_place,
594                killed, killed_cnt,
595                lost_tasks, lost_f_set, lost_lf_set,
596                rec_child, rec_to, rec_from, rec_from_waiting⟩

598 Store_ReceivingTerminateGhostSignal ≜
599   ∧ exec_state = "running"
600   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("rf", "terminateGhost")
601         rf ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
602              ELSE C!FindResilientFinishById(msg.finish_id)
603         ghost_child ≜ msg.ghost_finish_id
604         rf_updated ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
605                      ELSE [rf EXCEPT !.ghost_children =
606                                      rf.ghost_children \ {ghost_child}]
607     IN  ∧ msg ≠ C!NOT_MESSAGE
608         ∧ ¬C!IsRecoveringTasksToDeadPlaces(rf.id)
609         ∧ C!ApplyTerminateSignal(rf, rf_updated, msg)
610         ∧ step' = step + 1
611   ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
612                nxt_finish_id, nxt_task_id, nxt_remote_place,
613                killed, killed_cnt,
614                lost_tasks, lost_f_set, lost_lf_set,
615                rec_child, rec_to, rec_from, rec_from_waiting⟩

617 Store_FindingGhostChildren ≜
618   ∧ exec_state = "running"
619   ∧ LET req ≜ C!FindMarkGhostChildrenRequest
620         rf ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
621              ELSE C!FindResilientFinishById(req.finish_id)
622         ghosts ≜ IF req = C!NOT_REQUEST THEN {}
623                  ELSE C!GetNonAdoptedGhostChildren(rf.id)
624         grf ≜ C!ChooseGhost(ghosts)
625         grf_updated ≜ IF grf = C!NOT_FINISH THEN C!NOT_FINISH
626                       ELSE [grf EXCEPT !.isAdopted = TRUE]
627         req_updated ≜ IF req = C!NOT_REQUEST THEN C!NOT_REQUEST
628                       ELSE [req EXCEPT !.markingDone = TRUE]
629     IN  ∧ req ≠ C!NOT_REQUEST
630         ∧ rf ≠ C!NOT_FINISH
631         ∧ IF ghosts = {}
632           THEN ∧ rf_set' = rf_set
633                ∧ rec_child' = (rec_child \ {req}) ∪ {req_updated}
```

```
                ELSE  ∧ rf_set' = (rf_set \ {grf}) ∪ {grf_updated}
                      ∧ rec_child' = rec_child
            ∧ step' = step + 1
    ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set, msgs,
                 nxt_finish_id, nxt_task_id, nxt_remote_place,
                 killed, killed_cnt,
                 lost_tasks, lost_f_set, lost_lf_set,
                 rec_to, rec_from, rec_from_waiting⟩

Store_AddingGhostChildren ≜
    ∧ exec_state = "running"
    ∧ LET req ≜ C!FindAddGhostChildrenRequest
          rf ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
               ELSE C!FindResilientFinishById(req.finish_id)
          ghosts ≜ C!GetAdoptedGhostChildren(rf.id)
          rf_updated ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
                       ELSE [rf EXCEPT !.ghost_children =
                                       rf.ghost_children ∪ ghosts]
      IN  ∧ req ≠ C!NOT_REQUEST
          ∧ rf ≠ C!NOT_FINISH
          ∧ rf_set' = (rf_set \ {rf}) ∪ {rf_updated}
          ∧ rec_child' = rec_child \ {req}
          ∧ step' = step + 1
    ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set, msgs,
                 nxt_finish_id, nxt_task_id, nxt_remote_place,
                 killed, killed_cnt,
                 lost_tasks, lost_f_set, lost_lf_set,
                 rec_to, rec_from, rec_from_waiting⟩

Store_CancellingTasksToDeadPlace ≜
    ∧ exec_state = "running"
    ∧ LET req ≜ C!FindToDeadRequest
          rf ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
               ELSE C!FindResilientFinishById(req.finish_id)
          rf_updated ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
                       ELSE [rf EXCEPT !.transOrLive[req.from][req.to] = 0,
                                       !.gc = rf.gc - rf.transOrLive[req.from][req.to]]
      IN  ∧ req ≠ C!NOT_REQUEST
          ∧ rf ≠ C!NOT_FINISH
          ∧ C!ApplyTerminateSignal2(rf, rf_updated)
          ∧ rec_to' = rec_to \ {req}
          ∧ step' = step + 1
    ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
                 nxt_finish_id, nxt_task_id, nxt_remote_place,
                 killed, killed_cnt,
```

```
679                 lost_tasks, lost_f_set, lost_lf_set,
680                 rec_child, rec_from, rec_from_waiting⟩

682 Store_SendingCountTransitSignalToLocalFinish ≜
683   ∧ exec_state = "running"
684   ∧ LET req ≜ C!FindFromDeadRequest
685         rf ≜ IF req = C!NOT_REQUEST THEN C!NOT_FINISH
686              ELSE IF ¬C!ResilientFinishExists(req.finish_id)
687              THEN C!NOT_FINISH
688              ELSE C!FindResilientFinishById(req.finish_id)
689         msg ≜ IF req = C!NOT_REQUEST THEN C!NOT_MESSAGE
690               ELSE [from ↦ "rf", to ↦ "dst", tag ↦ "countDropped",
691                     finish_id ↦ rf.id,
692                     src ↦ req.from, dst ↦ req.to,
693                     num_sent ↦ rf.sent[req.from][req.to]]
694     IN  ∧ req ≠ C!NOT_REQUEST
695         ∧ rec_from' = rec_from \ {req}
696         ∧ IF rf ≠ C!NOT_FINISH
697           THEN ∧ C!SendMsg(msg)
698                ∧ rec_from_waiting' = rec_from_waiting ∪ {req}
699           ELSE ∧ msgs' = msgs  resilient finish has been released already
700                ∧ rec_from_waiting' = rec_from_waiting
701         ∧ step' = step + 1
702   ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set, rf_set,
703                nxt_finish_id, nxt_task_id, nxt_remote_place,
704                killed, killed_cnt,
705                lost_tasks, lost_f_set, lost_lf_set,
706                rec_child, rec_to⟩

708 Store_CancellingTransitTasksFromDeadPlace ≜
709   ∧ exec_state = "running"
710   ∧ LET msg ≜ C!FindMessageToActivePlaceWithTag("rf", "countDroppedDone")
711         from ≜ msg.src
712         to ≜ msg.dst
713         finish_id ≜ msg.finish_id
714         req ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_REQUEST
715               ELSE C!FindFromDeadWaitingRequest(finish_id, from, to)
716         rf ≜ IF msg = C!NOT_MESSAGE THEN C!NOT_FINISH
717              ELSE IF ¬C!ResilientFinishExists(req.finish_id) THEN C!NOT_FINISH
718              ELSE C!FindResilientFinishById(finish_id)
719         rf_updated ≜ IF rf = C!NOT_FINISH THEN C!NOT_FINISH
720                      ELSE [rf EXCEPT
721                       !.transOrLive[from][to] =
722                         rf.transOrLive[from][to] − msg.num_dropped,
723                       !.gc = rf.gc − msg.num_dropped]
```

```
724     IN  ∧ msg ≠ C!NOT_MESSAGE
725         ∧ rec_from_waiting' = rec_from_waiting \ {req}
726         ∧ IF msg.num_dropped > 0
727           THEN C!ApplyTerminateSignal(rf, rf_updated, msg)
728           ELSE C!RecvCountDroppedResponse(msg)
729         ∧ step' = step + 1
730   ∧ UNCHANGED ⟨exec_state, tasks, f_set, lf_set,
731                nxt_finish_id, nxt_task_id, nxt_remote_place,
732                killed, killed_cnt,
733                lost_tasks, lost_f_set, lost_lf_set,
734                rec_child, rec_to, rec_from⟩

736 ──────────────────────────────────────────────────────────────
737 KillingPlace ≜
738   ∧ exec_state = "running"
739   ∧ killed_cnt < MAX_KILL
740   ∧ LET victim ≜ CHOOSE x ∈ (C!PlaceID \ killed ) : x ≠ 0
741         victim_tasks ≜ C!FindLostTasks(victim)
742         victim_finishes ≜ C!FindLostFinishes(victim)
743         victim_local_finishes ≜ C!FindLostLocalFinishes(victim)
744         rf_to ≜ C!FindImpactedResilientFinishToDead(victim)
745         rf_from ≜ C!FindImpactedResilientFinishFromDead(victim)
746     IN  ∧ step ≥ KILL_FROM
747         ∧ step < KILL_TO
748         ∧ killed' = killed ∪ {victim}
749         ∧ killed_cnt' = killed_cnt + 1
750         ∧ lost_tasks' = lost_tasks ∪ victim_tasks
751         ∧ tasks' = tasks \ victim_tasks
752         ∧ lost_f_set' = lost_f_set ∪ victim_finishes
753         ∧ f_set' = f_set \ victim_finishes
754         ∧ lost_lf_set' = lost_lf_set ∪ victim_local_finishes
755         ∧ lf_set' = lf_set \ victim_local_finishes
756         ∧ rec_child' = rec_child ∪ {
757                 task ∈ C!GetChildrenTask : ∧ task.finish_id ∈ rf_to
758                                            ∧ task.victim = victim
759                                            ∧ task.markingDone = FALSE
760                                          }
761         ∧ rec_to' = rec_to ∪ {
762                   task ∈ C!ConvTask : ∃ rf ∈ rf_set : ∃ p ∈ C!PlaceID :
763                   ∧ task.finish_id = rf.id
764                   ∧ task.finish_id ∈ rf_to
765                   ∧ rf.transOrLive[p][victim] > 0
766                   ∧ task.from = p
767                   ∧ task.to = victim
```

```
                                    }
        ∧ rec_from' = rec_from ∪ {
                        task ∈ C!ConvTask : ∃ rf ∈ rf_set : ∃ p ∈ C!PlaceID :
                         ∧ task.finish_id = rf.id
                         ∧ task.finish_id ∈ rf_to
                         ∧ rf.transOrLive[victim][p] > 0
                         ∧ task.to = p
                         ∧ task.from = victim
                                    }
        ∧ step' = step + 1
  ∧ UNCHANGED ⟨exec_state, rf_set, msgs,
                nxt_finish_id, nxt_task_id, nxt_remote_place,
                rec_from_waiting⟩

Program_Terminating ≜
  ∧ exec_state = "running"
  ∧ LET root_task ≜ CHOOSE task ∈ tasks : task.id = C!ROOT_TASK_ID
        root_task_updated ≜ [root_task EXCEPT !.status = "terminated"]
    IN  ∧ root_task.status = "running"  root task unblocked
        ∧ tasks' = (tasks \ {root_task}) ∪ {root_task_updated}
        ∧ exec_state' = "success"
        ∧ step' = step + 1
  ∧ UNCHANGED ⟨f_set, lf_set, rf_set, msgs,
                nxt_finish_id, nxt_task_id, nxt_remote_place,
                killed, killed_cnt,
                lost_tasks, lost_f_set, lost_lf_set,
                rec_child, rec_to, rec_from, rec_from_waiting⟩

Possible next actions at each state

Next ≜
  ∨ Task_CreatingFinish
  ∨ Finish_CreatingRemoteTask
  ∨ Finish_TerminatingTask
  ∨ Finish_ReceivingPublishDoneSignal
  ∨ Finish_ReceivingReleaseSignal
  ∨ LocalFinish_CreatingRemoteTask
  ∨ LocalFinish_TerminatingTask
  ∨ LocalFinish_MarkingDeadPlace
  ∨ SendingTask
  ∨ DroppingTask
  ∨ ReceivingTask
  ∨ Store_ReceivingPublishSignal
  ∨ Store_ReceivingTransitSignal
```

$$\lor\ Store\_ReceivingTerminateTaskSignal$$
$$\lor\ Store\_ReceivingTerminateGhostSignal$$
$$\lor\ Store\_FindingGhostChildren$$
$$\lor\ Store\_AddingGhostChildren$$
$$\lor\ Store\_CancellingTasksToDeadPlace$$
$$\lor\ Store\_SendingCountTransitSignalToLocalFinish$$
$$\lor\ Store\_CancellingTransitTasksFromDeadPlace$$
$$\lor\ KillingPlace$$
$$\lor\ Program\_Terminating$$

We assume weak fairness on all actions (*i.e.* an action that remains forever enabled, must eventually be executed).

$$Liveness \triangleq$$
$$\land \mathrm{WF}_{Vars}(Task\_CreatingFinish)$$
$$\land \mathrm{WF}_{Vars}(Finish\_CreatingRemoteTask)$$
$$\land \mathrm{WF}_{Vars}(Finish\_TerminatingTask)$$
$$\land \mathrm{WF}_{Vars}(Finish\_ReceivingPublishDoneSignal)$$
$$\land \mathrm{WF}_{Vars}(Finish\_ReceivingReleaseSignal)$$
$$\land \mathrm{WF}_{Vars}(LocalFinish\_CreatingRemoteTask)$$
$$\land \mathrm{WF}_{Vars}(LocalFinish\_TerminatingTask)$$
$$\land \mathrm{WF}_{Vars}(LocalFinish\_MarkingDeadPlace)$$
$$\land \mathrm{WF}_{Vars}(SendingTask)$$
$$\land \mathrm{WF}_{Vars}(DroppingTask)$$
$$\land \mathrm{WF}_{Vars}(ReceivingTask)$$
$$\land \mathrm{WF}_{Vars}(Store\_ReceivingPublishSignal)$$
$$\land \mathrm{WF}_{Vars}(Store\_ReceivingTransitSignal)$$
$$\land \mathrm{WF}_{Vars}(Store\_ReceivingTerminateTaskSignal)$$
$$\land \mathrm{WF}_{Vars}(Store\_ReceivingTerminateGhostSignal)$$
$$\land \mathrm{WF}_{Vars}(Store\_FindingGhostChildren)$$
$$\land \mathrm{WF}_{Vars}(Store\_AddingGhostChildren)$$
$$\land \mathrm{WF}_{Vars}(Store\_CancellingTasksToDeadPlace)$$
$$\land \mathrm{WF}_{Vars}(Store\_SendingCountTransitSignalToLocalFinish)$$
$$\land \mathrm{WF}_{Vars}(Store\_CancellingTransitTasksFromDeadPlace)$$
$$\land \mathrm{WF}_{Vars}(KillingPlace)$$
$$\land \mathrm{WF}_{Vars}(Program\_Terminating)$$

Specification

$$Spec \triangleq \quad Init \land \Box[Next]_{Vars} \land Liveness$$

THEOREM $Spec \implies \Box(TypeOK)$

```
1  ---------- MODULE OptimisticCommons ----------
2  Constants and common utility actions
3  EXTENDS Integers

5  CONSTANTS LEVEL,
6            WIDTH,
7            NUM_PLACES,
8            MAX_KILL

10 VARIABLES exec_state,
11           tasks,
12           f_set,
13           lf_set,
14           rf_set,
15           msgs,
16           nxt_finish_id,
17           nxt_task_id,
18           nxt_remote_place,
19           killed,
20           rec_child,
21           rec_to,
22           rec_from,
23           rec_from_waiting

25 ----------------------------------------------

27 FIRST_PLACE_ID ≜ 0
28 PlaceID ≜ FIRST_PLACE_ID .. (NUM_PLACES − 1)
29 NOT_PLACE_ID ≜ −1

31 ROOT_FINISH_ID ≜ 0
32 MAX_FINISH_ID ≜ ((1 − (WIDTH)^(LEVEL+1)) ÷ (1 − WIDTH))  the power series
33 NOT_FINISH_ID ≜ −1
34 FinishID ≜ ROOT_FINISH_ID .. MAX_FINISH_ID

36 ROOT_TASK_ID ≜ 0
37 MAX_TASK_ID ≜ MAX_FINISH_ID
38 NOT_TASK_ID ≜ −1
39 TaskID ≜ ROOT_TASK_ID .. MAX_TASK_ID

41 BranchID ≜ 0 .. WIDTH
42 LevelID ≜ 0 .. LEVEL

44 TASK_STATUS ≜ {"waitingForPublish", "waitingForTransit", "sent", "dropped",
45                "running", "blocked", "terminated"}

47 TASK_TYPE ≜ {"normal",  terminates at any time
48              "finishMainTask"  terminates after finish creates all its branches
```

```
49                  }

51  FINISH_STATUS ≜ {"active", "waitingForPublish", "waitingForRelease", "released"}

53  Task ≜ [id : TaskID,
54          pred_id : TaskID ∪ {NOT_TASK_ID},   predecessor task, used for debugging only
55          src : PlaceID, dst : PlaceID,
56          finish_id : FinishID,
57          level : LevelID,
58          last_branch : BranchID,
59          status : TASK_STATUS,
60          type : TASK_TYPE,
61          finish_type : {"global", "local", "N/A"}]

63  RootTask ≜ [id ↦ ROOT_TASK_ID,
64              pred_id ↦ NOT_TASK_ID,
65              src ↦ FIRST_PLACE_ID,
66              dst ↦ FIRST_PLACE_ID,
67              finish_id ↦ ROOT_FINISH_ID,
68              level ↦ 0,
69              last_branch ↦ WIDTH,
70              status ↦ "blocked",
71              type ↦ "normal",
72              finish_type ↦ "global"]

74  NOT_TASK ≜ [id ↦ NOT_TASK_ID,
75              src ↦ NOT_PLACE_ID,
76              dst ↦ NOT_PLACE_ID,
77              level ↦ −1,
78              finish_id ↦ NOT_FINISH_ID,
79              finish_type ↦ "N/A"]

82  Place1D ≜ [PlaceID → Nat]
83  Place2D ≜ [PlaceID → [PlaceID → Nat]]

85  Place1DZeros ≜ [i ∈ PlaceID ↦ 0]
86  Place2DZeros ≜ [i ∈ PlaceID ↦ [j ∈ PlaceID ↦ 0]]

88  Place1DTerminateTask(src, cnt) ≜ [i ∈ PlaceID ↦
89                                    IF i = src THEN cnt ELSE 0]

91  Place2DInitResilientFinish(home) ≜ [i ∈ PlaceID ↦ [j ∈ PlaceID ↦
92                                    IF i = home ∧ j = home THEN 1 ELSE 0]]

94  Finish ≜ [id : FinishID \ {ROOT_FINISH_ID},
95            pred_id : TaskID ∪ {NOT_TASK_ID},   predecessor task
96            home : PlaceID,
```

```
97              origin : PlaceID,
98              parent_finish_id : FinishID,
99              status : FINISH_STATUS,
100             lc : Nat]

102 RootFinish ≜ [id ↦ ROOT_FINISH_ID + 1,
103               pred_id ↦ RootTask.id,
104               home ↦ FIRST_PLACE_ID,
105               origin ↦ FIRST_PLACE_ID,
106               parent_finish_id ↦ ROOT_FINISH_ID,
107               status ↦ "active",
108               lc ↦ 1]

110 RootFinishTask ≜ [id ↦ ROOT_TASK_ID + 1,
111                   pred_id ↦ ROOT_TASK_ID,
112                   dst ↦ FIRST_PLACE_ID,
113                   src ↦ FIRST_PLACE_ID,
114                   finish_id ↦ RootFinish.id,
115                   status ↦ "running",
116                   level ↦ 1,
117                   last_branch ↦ 0,
118                   type ↦ "finishMainTask",
119                   finish_type ↦ "global"]

121 NOT_FINISH ≜ [id ↦ NOT_FINISH_ID,
122                home ↦ NOT_PLACE_ID,
123                origin ↦ NOT_PLACE_ID,
124                parent_finish_id ↦ NOT_FINISH_ID,
125                status ↦ "",
126                lc ↦ 0]

128 LFinish ≜ [id : FinishID \ {ROOT_FINISH_ID},
129           home : PlaceID,
130           lc : Nat,
131           reported : Place1D,
132           received : Place1D,
133           deny : Place1D]

135 RFinish ≜ [id : FinishID \ {ROOT_FINISH_ID},
136           home : PlaceID,
137           origin : PlaceID,
138           parent_finish_id : FinishID,
139           transOrLive : Place2D,
140           sent : Place2D,
141           gc : Nat,
142           ghost_children : SUBSET FinishID,
```

```
143             isAdopted : BOOLEAN ]

145 Message ≜ [from : {"f", "rf", "src", "dst", "lf"},
146             to : {"f", "rf", "src", "dst", "lf"},
147             tag : {"transit", "transitDone", "transitNotDone",
148                    "terminateTask", "terminateGhost",
149                    "task",
150                    "publish", "publishDone",
151                    "release",
152                    "countDropped", "countDroppedDone"},
153             src : PlaceID,
154             dst : PlaceID,
155             finish_id : FinishID,
156             ghost_finish_id : FinishID,
157             task_id : TaskID,
158             term_tasks_by_src : Place1D,   termination only
159             term_tasks_dst : PlaceID,     termination only
160             num_sent : Nat,
161             num_dropped : Nat
162           ]

164 NOT_MESSAGE ≜ [from ↦ "N/A", to ↦ "N/A", tag ↦ "N/A",
165                 src ↦ NOT_PLACE_ID, dst ↦ NOT_PLACE_ID,
166                 finish_id ↦ NOT_FINISH_ID,
167                 task_id ↦ NOT_TASK_ID,
168                 ghost_finish_id ↦ NOT_FINISH_ID,
169                 term_tasks_by_src ↦ Place1DZeros,
170                 term_tasks_dst ↦ NOT_PLACE_ID]

172 FindRunningTask(maxLevel) ≜
173   LET tset ≜ {task ∈ tasks : ∧ task.status = "running"
174                              ∧ task.last_branch < WIDTH
175                              ∧ task.level ≤ maxLevel}
176   IN  IF tset = {} THEN NOT_TASK
177       ELSE CHOOSE t ∈ tset : TRUE

179 FindRunningTaskWithFinishType(maxLevel, fin_type) ≜
180   LET tset ≜ {task ∈ tasks : ∧ task.status = "running"
181                              ∧ task.last_branch < WIDTH
182                              ∧ task.level ≤ maxLevel
183                              ∧ task.finish_type = fin_type}
184   IN  IF tset = {} THEN NOT_TASK
185       ELSE CHOOSE t ∈ tset : TRUE

187 FindFinishById(id) ≜
188   CHOOSE f ∈ f_set : f.id = id
```

```
FindResilientFinishById(id) ≜
  CHOOSE f ∈ rf_set : f.id = id

FindTaskById(id) ≜
  CHOOSE t ∈ tasks : t.id = id

ActiveFinishSet ≜ {"active", "waitingForRelease"}

FindActiveFinish(id, home) ≜
  LET fset ≜ {finish ∈ f_set : ∧ finish.status ∈ ActiveFinishSet
                               ∧ finish.id = id
                               ∧ ∨ ∧ home ≠ NOT_PLACE_ID
                                   ∧ finish.home = home
                                 ∨ ∧ home = NOT_PLACE_ID}
  IN  IF fset = {} THEN NOT_FINISH
      ELSE CHOOSE f ∈ fset : TRUE

FindPendingRemoteTask(finish_id, status) ≜
  LET tset ≜ {task ∈ tasks : ∧ task.status = status
                             ∧ task.src ≠ task.dst
                             ∧ task.finish_type = "local"
                             ∧ task.finish_id = finish_id
             }
  IN  IF tset = {} THEN NOT_TASK
      ELSE CHOOSE t ∈ tset : TRUE

IsPublished(finish_id) ≜
  ∃ rf ∈ rf_set : ∧ rf.id = finish_id
                  ∧ ∃ f ∈ f_set : ∧ f.id = finish_id
                                  ∧ f.status ∈ ActiveFinishSet
LocalFinishExists(place, finish_id) ≜
  ∃ lf ∈ lf_set : ∧ lf.id = finish_id
                  ∧ lf.home = place

ResilientFinishExists(finish_id) ≜
  ∃ rf ∈ rf_set : rf.id = finish_id

FindLocalFinish(place, finish_id) ≜
  CHOOSE f ∈ lf_set : f.home = place ∧ f.id = finish_id

FindFinishToRelease(finish_id) ≜
  CHOOSE f ∈ f_set : f.id = finish_id ∧ f.status = "waitingForRelease" ∧ f.lc = 0

 a task can terminate if cannot branch further - at last level or at last branch number
FindTaskToTerminate(fin_type) ≜
  LET tset ≜ {task ∈ tasks : ∧ task.status = "running"
                             ∧ task.finish_type = fin_type
                             ∧ ∨ task.level = LEVEL
```

```
238                                   ∨ task.last_branch = WIDTH
239                                ∧ IF fin_type = "global"
240                                   THEN FindActiveFinish(task.finish_id, task.src)
241                                          ≠ NOT_FINISH
242                                   ELSE TRUE
243              }
244   IN  IF tset = {} THEN NOT_TASK
245       ELSE CHOOSE t ∈ tset : TRUE

247 FindBlockedTask(task_id) ≜
248   LET tset ≜ {task ∈ tasks : ∧ task.status = "blocked"
249                              ∧ task.id = task_id
250              }
251   IN  IF tset = {} THEN NOT_TASK
252       ELSE CHOOSE t ∈ tset : TRUE

254 SendMsg(m) ≜
    Add message to the msgs set
258   msgs' = msgs ∪ {m}

260 RecvMsg(m) ≜
    Delete message from the msgs set
264   msgs' = msgs \ {m}

266 ReplaceMsg(toRemove, toAdd) ≜
    Remove an existing message and add another one
270   msgs' = (msgs \ {toRemove}) ∪ {toAdd}

272 FindMessageToActivePlaceWithTag(to, tag) ≜
    Return a message matching the given criteria, or NOT_MESSAGE otherwise
276   LET mset ≜ {m ∈ msgs : ∧ m.to = to
277                          ∧ m.tag = tag
278                          ∧ IF m.to = "rf"
279                            THEN TRUE
280                            ELSE m.dst ∉ killed}
281   IN  IF mset = {} THEN NOT_MESSAGE
282       ELSE (CHOOSE x ∈ mset : TRUE)

284 Sum(place1D) ≜
285     IF NUM_PLACES = 0 THEN 0
286      ELSE IF NUM_PLACES = 1 THEN place1D[0]
287      ELSE IF NUM_PLACES = 2 THEN place1D[0] + place1D[1]
288      ELSE IF NUM_PLACES = 3 THEN place1D[0] + place1D[1] + place1D[2]
289      ELSE IF NUM_PLACES = 4 THEN place1D[0] + place1D[1] + place1D[2]
290                                    + place1D[3]
291      ELSE IF NUM_PLACES = 5 THEN place1D[0] + place1D[1] + place1D[2]
```

```
                                        +  place1D[3] + place1D[4]
    ELSE IF NUM_PLACES = 6 THEN place1D[0] +  place1D[1] + place1D[2]
                                        +  place1D[3] + place1D[4] + place1D[5]
    ELSE  -1

NextRemotePlace(here) ≜
 IF nxt_remote_place[here] = here
  THEN (nxt_remote_place[here] + 1)%NUM_PLACES
  ELSE  nxt_remote_place[here]

ShiftNextRemotePlace(here) ≜
 IF nxt_remote_place[here] = here
  THEN nxt_remote_place' = [nxt_remote_place EXCEPT ![here] =
                                (nxt_remote_place[here] + 2)%NUM_PLACES]
  ELSE  nxt_remote_place' = [nxt_remote_place EXCEPT ![here] =
                                (nxt_remote_place[here] + 1)%NUM_PLACES]
----------------------------------------------------------------------
FindLostTasks(dead) ≜
 {
   task ∈ tasks : ∨ ∧ task.status ∈ {"waitingForPublish",
                                     "waitingForTransit"}
                  ∧ task.src = dead
                ∨ ∧ task.status ∈ {"running", "blocked"}
                  ∧ task.dst = dead
 }

FindLostFinishes(dead) ≜
 {
   finish ∈ f_set : ∧ finish.status ≠ "released"
                    ∧ finish.home = dead
 }

FindLostLocalFinishes(dead) ≜
 {
   local_fin ∈ lf_set : ∧ local_fin.home = dead
 }

FindImpactedResilientFinish(victim) ≜
{
 id ∈ FinishID : ∃ rf ∈ rf_set : ∧ rf.id = id
                                 ∧ ∨ ∃ i ∈ PlaceID
                                     : rf.transOrLive[i][victim] > 0
                                   ∨ ∃ j ∈ PlaceID
                                     : rf.transOrLive[victim][j] > 0
}
```

```
338 FindImpactedResilientFinishToDead(victim) ≜
339 {
340   id ∈ FinishID : ∃ rf ∈ rf_set : ∧ rf.id = id
341                                   ∧ ∃ i ∈ PlaceID
342                                       : rf.transOrLive[i][victim] > 0
343 }

345 FindImpactedResilientFinishFromDead(victim) ≜
346 {
347   id ∈ FinishID : ∃ rf ∈ rf_set : ∧ rf.id = id
348                                   ∧ ∃ j ∈ PlaceID
349                                       : rf.transOrLive[victim][j] > 0
350 }

352 GetSrc(rf) ≜
353   CHOOSE i ∈ PlaceID : CHOOSE j ∈ killed : rf.transOrLive[i][j] > 0

355 GetDst(rf, src) ≜
356   CHOOSE j ∈ killed : rf.transOrLive[src][j] > 0

358 GetAdoptedGhostChildren(fin_id) ≜
359  {
360    id ∈ FinishID : ∃ rf ∈ rf_set : ∧ rf.id = id
361                                    ∧ rf.home ∈ killed
362                                    ∧ rf.parent_finish_id = fin_id
363                                    ∧ rf.isAdopted = TRUE
364  }

366 GetNonAdoptedGhostChildren(fin_id) ≜
367  {
368    id ∈ FinishID : ∃ rf ∈ rf_set : ∧ rf.id = id
369                                    ∧ rf.home ∈ killed
370                                    ∧ rf.parent_finish_id = fin_id
371                                    ∧ rf.isAdopted = FALSE
372  }

374 IsRecoveringTasksToDeadPlaces(fin_id) ≜
375   ∨ ∃ task ∈ rec_child : task.finish_id = fin_id
376   ∨ ∃ task ∈ rec_to    : task.finish_id = fin_id

378 ConvTask ≜ [finish_id : FinishID, from : PlaceID, to : PlaceID]

380 GetChildrenTask ≜ [finish_id : FinishID, victim : PlaceID, markingDone : BOOLEAN ]

382 NOT_REQUEST ≜ [finish_id ↦ NOT_FINISH_ID]

384 ChildRequestExists(fin_id) ≜
385   ∃ creq ∈ rec_child : fin_id = creq.finish_id
```

```
387 ToRequestExists(fin_id) ≜
388   ∃ treq ∈ rec_to : fin_id = treq.finish_id

390 FindMarkGhostChildrenRequest ≜
391   LET rset ≜ {r ∈ rec_child : r.markingDone = FALSE}
392   IN  IF rset = {} THEN NOT_REQUEST
393       ELSE (CHOOSE x ∈ rset : TRUE)

395 FindAddGhostChildrenRequest ≜
396   LET rset ≜ {r ∈ rec_child : r.markingDone = TRUE}
397   IN  IF rset = {} THEN NOT_REQUEST
398       ELSE (CHOOSE x ∈ rset : TRUE)

400 ChooseGhost(ghosts) ≜
401   IF ghosts = {} THEN NOT_FINISH ELSE CHOOSE x ∈ rf_set : x.id ∈ ghosts

403 FindToDeadRequest ≜
404   IF rec_to = {} THEN NOT_REQUEST
405    ELSE IF ∃ a ∈ rec_to : ¬ChildRequestExists(a.finish_id)
406         THEN (CHOOSE b ∈ rec_to : ¬ChildRequestExists(b.finish_id))
407         ELSE NOT_REQUEST

409 FindFromDeadRequest ≜
410   IF rec_from = {} THEN NOT_REQUEST
411    ELSE IF ∃ a ∈ rec_from : ∧ ¬ChildRequestExists(a.finish_id)
412                             ∧ ¬ToRequestExists(a.finish_id)
413         THEN (CHOOSE b ∈ rec_from : ∧ ¬ChildRequestExists(b.finish_id)
414                                     ∧ ¬ToRequestExists(b.finish_id))
415         ELSE NOT_REQUEST

417 FindFromDeadWaitingRequest(fin_id, from, to) ≜
418   CHOOSE x ∈ rec_from_waiting : ∧ x.finish_id = fin_id
419                                 ∧ x.from = from
420                                 ∧ x.to = to

422 ApplyTerminateSignal(rf, rf_updated, msg) ≜
423   IF rf_updated.gc = 0 ∧ rf_updated.ghost_children = {}
424    THEN IF rf.isAdopted
425         THEN ∧ ReplaceMsg(msg, [from ↦ "rf", to ↦ "rf", tag ↦ "terminateGhost",
426                                 finish_id ↦ rf.parent_finish_id,
427                                 ghost_finish_id ↦ rf.id,
428                                 dst ↦ NOT_PLACE_ID]) rf.id is enough
429              ∧ rf_set' = rf_set \ {rf}
430         ELSE ∧ ReplaceMsg(msg, [from ↦ "rf", to ↦ "f", tag ↦ "release",
431                                 finish_id ↦ rf.id,
432                                 dst ↦ rf.home])
433              ∧ rf_set' = rf_set \ {rf}
```

```
434     ELSE  ∧ RecvMsg(msg)
435           ∧ rf_set' = (rf_set \ {rf}) ∪ {rf_updated}

437 ApplyTerminateSignal2(rf, rf_updated) ≜
438   IF rf_updated.gc = 0 ∧ rf_updated.ghost_children = {}
439    THEN IF rf.isAdopted
440           THEN ∧ SendMsg([from ↦ "rf", to ↦ "rf", tag ↦ "terminateGhost",
441                           finish_id ↦ rf.parent_finish_id,
442                           ghost_finish_id ↦ rf.id,
443                           dst ↦ NOT_PLACE_ID])  rf.id is enough
444                ∧ rf_set' = rf_set \ {rf}
445           ELSE ∧ SendMsg([from ↦ "rf", to ↦ "f", tag ↦ "release",
446                           finish_id ↦ rf.id,
447                           dst ↦ rf.home])
448                ∧ rf_set' = rf_set \ {rf}
449    ELSE  ∧ msgs' = msgs
450          ∧ rf_set' = (rf_set \ {rf}) ∪ {rf_updated}

452 RecvTerminateSignal(msg) ≜
453   ∧ RecvMsg(msg)
454   ∧ rf_set' = rf_set

456 RecvCountDroppedResponse(msg) ≜
457    ∧ RecvMsg(msg)
458    ∧ rf_set' = rf_set
459
```

Appendix C

TLA+ Specification of the Distributed Finish Replication Protocol

```
1  ---------- MODULE AsyncFinishReplication ----------
2  EXTENDS Integers

4  CONSTANTS CLIENT_NUM,   the number of clients
5            MAX_KILL      maximum allowed kill events

7  VARIABLES exec_state,    the execution state of the program: running, success, or fatal
8            clients,      clients sending value update requests to master and backup
9            master,       array of master instances, only one is active
10           backup,       array of backup instances, only one is active
11           msgs,         in-flight messages
12           killed        number of invoked kill actions to master or backup

14 ---------------------------------------------------
15 Vars ≜ ⟨exec_state, clients, master, backup, msgs, killed⟩
16 ---------------------------------------------------
17 C ≜ INSTANCE Commons
18 ---------------------------------------------------
19 TypeOK ≜
     Variables type constrains
23   ∧ clients ∈ [C!CLIENT_ID → C!Client]
24   ∧ master ∈ [C!INSTANCE_ID → C!Master]
25   ∧ backup ∈ [C!INSTANCE_ID → C!Backup]
26   ∧ exec_state ∈ {"running", "success", "fatal"}
27   ∧ msgs ⊆ C!Messages
28   ∧ killed ∈ 0 .. MAX_KILL

30 ---------------------------------------------------
```

```
31 MaxOneActiveMaster ≜
   Return true if maximum one active master exists, and false otherwise
35   LET activeM ≜ C!FindMaster(C!INST_STATUS_ACTIVE)
36       otherIds ≜ C!INSTANCE_ID \ {activeM.id}
37   IN  IF activeM = C!NOT_MASTER
38       THEN TRUE  zero active masters
39       ELSE  LET otherActiveMs ≜ {m ∈ otherIds :
40                                   master[m].status = C!INST_STATUS_ACTIVE}
41             IN  IF otherActiveMs = {} THEN TRUE  no other active masters
42                 ELSE  FALSE  other active masters exist

44 MaxOneActiveBackup ≜
   Return true if maximum one active backup exists, and false otherwise
48   LET activeB ≜ C!FindBackup(C!INST_STATUS_ACTIVE)
49       otherIds ≜ C!INSTANCE_ID \ {activeB.id}
50   IN  IF activeB = C!NOT_BACKUP
51       THEN TRUE  zero active backups
52       ELSE  LET otherActiveBs ≜ {b ∈ otherIds :
53                                   backup[b].status = C!INST_STATUS_ACTIVE}
54             IN  IF otherActiveBs = {} THEN TRUE  no other active backups
55                 ELSE  FALSE  other active backup exist

57 StateOK ≜
   State invariants
   1. on successful termination: the final version equals CLIENT_NUM
   2. on fatal termination: there must be a client whose master is lost and whose backup is lost or is
    unknown
   3. before termination:
    a) master version ≥ backup version
    b) master and backup version should not exceed CLIENT_NUM
    c) maximum one active master and maximum one active backup
68   LET curMaster ≜ C!LastKnownMaster
69       curBackup ≜ C!LastKnownBackup
70   IN  IF exec_state = "success"
71       THEN ∧ curMaster.version = CLIENT_NUM
72            ∧ curBackup.version = CLIENT_NUM
73       ELSE  IF exec_state = "fatal"
74       THEN ∃ c ∈ C!CLIENT_ID :
75              ∧ clients[c].phase = C!PH2_COMPLETED_FATAL
76              ∧ master[clients[c].masterId].status = C!INST_STATUS_LOST
77              ∧ IF clients[c].backupId ≠ C!UNKNOWN_ID
78                THEN backup[clients[c].backupId].status = C!INST_STATUS_LOST
79                ELSE  TRUE
80       ELSE  ∧ curMaster.version ≥ curBackup.version
81             ∧ curMaster.version ≤ CLIENT_NUM
82             ∧ curBackup.version ≤ CLIENT_NUM
```

```
83                ∧ MaxOneActiveMaster
84                ∧ MaxOneActiveBackup

86 ├──────────────────────────────────────────────────────────────┤
87 MustTerminate ≜
   Temporal property: the program must eventually terminate either successfully or fatally
92   ◇(exec_state ∈ {"success", "fatal"})
93 ├──────────────────────────────────────────────────────────────┤
94 Init ≜
   Initialize variables
98   ∧ exec_state = "running"
99   ∧ clients = [i ∈ C!CLIENT_ID ↦ [id ↦ i, phase ↦ C!PH1_PENDING,
100               value ↦ i,  masterId ↦ C!FIRST_ID,
101               backupId ↦ C!UNKNOWN_ID]]
102  ∧ backup = [i ∈ C!INSTANCE_ID ↦
103               IF i = C!FIRST_ID
104                THEN [id ↦ C!FIRST_ID, masterId ↦ C!FIRST_ID,
105                      status ↦ C!INST_STATUS_ACTIVE,
106                      value ↦ 0, version ↦ 0]
107                ELSE [id ↦ i, masterId ↦ C!UNKNOWN_ID,
108                      status ↦ C!INST_STATUS_NULL,
109                      value ↦ 0, version ↦ 0]]
110  ∧ master = [i ∈ C!INSTANCE_ID ↦
111               IF i = C!FIRST_ID
112                THEN [id ↦ C!FIRST_ID, backupId ↦ C!FIRST_ID,
113                      status ↦ C!INST_STATUS_ACTIVE,
114                      value ↦ 0, version ↦ 0]
115                ELSE [id ↦ i, backupId ↦ C!UNKNOWN_ID,
116                      status ↦ C!INST_STATUS_NULL,
117                      value ↦ 0, version ↦ 0]]
118  ∧ msgs = {}
119  ∧ killed = 0

121 ├──────────────────────────────────────────────────────────────┤
122 E_KillingMaster ≜
   Kill the active master instance.
126  ∧ exec_state = "running"
127  ∧ killed < MAX_KILL
128  ∧ LET activeM ≜ C!FindMaster(C!INST_STATUS_ACTIVE)
129    IN  ∧ activeM ≠ C!NOT_MASTER
130        ∧ master' = [master EXCEPT ![activeM.id].status = C!INST_STATUS_LOST]
131        ∧ killed' = killed + 1
132  ∧ UNCHANGED ⟨exec_state, clients, backup, msgs⟩

134 E_KillingBackup ≜
```

```
    Kill the active backup instance.
138  ∧ exec_state = "running"
139  ∧ killed < MAX_KILL
140  ∧ LET activeB ≜ C!FindBackup(C!INST_STATUS_ACTIVE)
141    IN ∧ activeB ≠ C!NOT_BACKUP
142       ∧ backup' = [backup EXCEPT ![activeB.id].status = C!INST_STATUS_LOST]
143       ∧ killed' = killed + 1
144  ∧ UNCHANGED ⟨exec_state, clients, master, msgs⟩

146 C_Starting ≜
    Client start the replication process by sending "do" to master
150  ∧ exec_state = "running"
151  ∧ LET client ≜ C!FindClient(C!PH1_PENDING)
152    IN ∧ client ≠ C!NOT_CLIENT
153       ∧ C!SendMsg([from ↦ "c",
154                    to ↦ "m",
155                    clientId ↦ client.id,
156                    masterId ↦ client.masterId,
157                    backupId ↦ C!UNKNOWN_ID,
158                    value ↦ client.value,
159                    tag ↦ "masterDo"])
160       ∧ clients' = [clients EXCEPT ![client.id].phase = C!PH2_WORKING]
161  ∧ UNCHANGED ⟨exec_state, master, backup, killed⟩

163 M_Doing ≜
    Master receiving "do", updating value, and sending "done"
167  ∧ exec_state = "running"
168  ∧ LET msg ≜ C!FindMessageToWithTag("m", C!INST_STATUS_ACTIVE, "masterDo")
169    IN ∧ msg ≠ C!NOT_MESSAGE
170       ∧ master' = [master EXCEPT ![msg.masterId].value =
171                                    master[msg.masterId].value + msg.value,
172                                  ![msg.masterId].version =
173                                    master[msg.masterId].version + 1]
174       ∧ C!ReplaceMsg(msg, [from ↦ "m",
175                            to ↦ "c",
176                            clientId ↦ msg.clientId,
177                            masterId ↦ msg.masterId,
178                            backupId ↦ master[msg.masterId].backupId,
179                            value ↦ 0,
180                            tag ↦ "masterDone"])
181  ∧ UNCHANGED ⟨exec_state, clients, backup, killed⟩

183 C_HandlingMasterDone ≜
    Client receiving "done" from master, and forwarding action to backup
```

```
187   ∧ exec_state = "running"
188   ∧ LET msg ≜ C!FindMessageToClient("m", "masterDone")
189     IN  ∧ msg ≠ C!NOT_MESSAGE
190         ∧ C!ReplaceMsg(msg, [from ↦ "c",
191                              to ↦ "b",
192                              clientId   ↦ msg.clientId,
193                              masterId ↦ msg.masterId,
194                              backupId ↦ msg.backupId,
195                              value ↦ clients[msg.clientId].value,
196                              tag ↦ "backupDo"])
197           update our knowledge about the backup identity
198         ∧ clients' = [clients EXCEPT ![msg.clientId].backupId = msg.backupId]
199   ∧ UNCHANGED ⟨exec_state, master, backup, killed⟩

201 B_Doing ≜
    Backup receiving "do", updating value, then sending "done"
205   ∧ exec_state = "running"
206   ∧ LET msg ≜ C!FindMessageToWithTag("b", C!INST_STATUS_ACTIVE, "backupDo")
207     IN  ∧ msg ≠ C!NOT_MESSAGE
208           Master info is consistent between client and backup
209         ∧ msg.masterId = backup[msg.backupId].masterId
210         ∧ backup' = [backup EXCEPT ![msg.backupId].value =
211                                      backup[msg.backupId].value + msg.value,
212                                    ![msg.backupId].version =
213                                      backup[msg.backupId].version + 1]
214         ∧ C!ReplaceMsg(msg, [from ↦ "b",
215                              to ↦ "c",
216                              clientId   ↦ msg.clientId,
217                              masterId ↦ msg.masterId,
218                              backupId ↦ msg.backupId,
219                              value ↦ 0,
220                              tag ↦ "backupDone"])
221   ∧ UNCHANGED ⟨exec_state, clients, master, killed⟩

223 B_DetectingOldMasterId ≜
    Backup receiving "do" and detecting that the client is using an old master id. It does not update the
    value, and it sends the new master id to the client
229   ∧ exec_state = "running"
230   ∧ LET msg ≜ C!FindMessageToWithTag("b", C!INST_STATUS_ACTIVE, "backupDo")
231     IN  ∧ msg ≠ C!NOT_MESSAGE
232           Master has changed, client must restart
233         ∧ msg.masterId ≠ backup[msg.backupId].masterId
234         ∧ C!ReplaceMsg(msg, [from ↦ "b",
235                              to ↦ "c",
236                              clientId   ↦ msg.clientId,
```

```
                                masterId ↦ backup[msg.backupId].masterId,
                                backupId ↦ msg.backupId,
                                value ↦ 0,
                                tag ↦ "newMasterId"])
  ∧ UNCHANGED ⟨exec_state, clients, master, backup, killed⟩

C_HandlingBackupDone ≜
  Client receiving "done" from backup. Replication completed
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToClient("b", "backupDone")
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ C!RecvMsg(msg)
        ∧ clients' = [clients EXCEPT ![msg.clientId].phase = C!PH2_COMPLETED]
        ∧ IF ∀ c ∈ C!CLIENT_ID : clients'[c].phase = C!PH2_COMPLETED
          THEN exec_state' = "success"
          ELSE exec_state' = exec_state
  ∧ UNCHANGED ⟨master, backup, killed⟩

C_HandlingMasterDoFailed ≜
  Client received the system's notification of a dead master, and is requesting the backup to return the new master info
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToWithTag("m", C!INST_STATUS_LOST, "masterDo")
        knownBackup ≜ IF msg ≠ C!NOT_MESSAGE
                      THEN C!FindBackup(C!INST_STATUS_ACTIVE)
                      ELSE C!NOT_BACKUP
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ IF knownBackup = C!NOT_BACKUP
          THEN ∧ C!RecvMsg(msg)
               ∧ exec_state' = "fatal"
               ∧ clients' = [clients EXCEPT ![msg.clientId].phase =
                                            C!PH2_COMPLETED_FATAL]
          ELSE ∧ C!ReplaceMsg(msg, [from ↦ "c",
                                    to ↦ "b",
                                    clientId ↦ msg.clientId,
                                    send the client's master knowledge,
                                    to force the backup to not respond
                                    until re-replication
                                    masterId ↦ clients[msg.clientId].masterId,
                                    backupId ↦ knownBackup.id,
                                    value ↦ 0,
                                    tag ↦ "backupGetNewMaster"])
               ∧ exec_state' = exec_state
```

```
286                     ∧ clients′ = clients
287     ∧ UNCHANGED ⟨master, backup, killed⟩

289 C_HandlingBackupDoFailed ≜
    Client received the system's notification of a dead backup, and is requesting the master to return the
    new backup info
294     ∧ exec_state = "running"
295     ∧ LET msg ≜ C!FindMessageToWithTag("b", C!INST_STATUS_LOST, "backupDo")
296       IN  ∧ msg ≠ C!NOT_MESSAGE
297           ∧ C!ReplaceMsg(msg, [from ↦ "c",
298                               to ↦ "m",
299                               clientId   ↦ msg.clientId,
300                               masterId ↦ clients[msg.clientId].masterId,
301                                send the client's backup knowledge,
302                                to force the master to not respond
303                                until rereplication
304                               backupId ↦ clients[msg.clientId].backupId,
305                               value ↦ 0,
306                               tag ↦ "masterGetNewBackup"])
307     ∧ UNCHANGED ⟨exec_state, clients, master, backup, killed⟩

309 ├──────────────────────────────────────────────────────────────────┤
310 M_GettingNewBackup ≜
    Master responding to client with updated backup identity
314     ∧ exec_state = "running"
315     ∧ LET msg ≜ C!FindMessageToWithTag("m",
316                                        C!INST_STATUS_ACTIVE,
317                                        "masterGetNewBackup")
318       IN  ∧ msg ≠ C!NOT_MESSAGE
319            master must not respond until it recovers the dead backup
320           ∧ msg.backupId ≠ master[msg.masterId].backupId
321           ∧ C!ReplaceMsg(msg, [from ↦ "m",
322                               to ↦ "c",
323                               clientId   ↦ msg.clientId,
324                               masterId ↦ msg.masterId,
325                               backupId ↦ master[msg.masterId].backupId,
326                               value ↦ 0,
327                               tag ↦ "newBackupId"])
328     ∧ UNCHANGED ⟨exec_state, clients, master, backup, killed⟩

330 B_GettingNewMaster ≜
    Backup responding to client with updated master identity
334     ∧ exec_state = "running"
335     ∧ LET msg ≜ C!FindMessageToWithTag("b",
336                                        C!INST_STATUS_ACTIVE,
```

```
                                                "backupGetNewMaster")
    IN  ∧ msg ≠ C!NOT_MESSAGE
          backup must not respond until it recovers the dead master
        ∧ msg.masterId ≠ backup[msg.backupId].masterId
        ∧ C!ReplaceMsg(msg, [from ↦ "b",
                             to ↦ "c",
                             clientId   ↦ msg.clientId,
                             masterId ↦ backup[msg.backupId].masterId,
                             backupId ↦ msg.backupId,
                             value ↦ 0,
                             tag ↦ "newMasterId"])
  ∧ UNCHANGED ⟨exec_state, clients, master, backup, killed⟩

C_HandlingBackupGetNewMasterFailed ≜
  The client handling the failure of the backup, when the client asked the backup to return the new
  master identity. The client mannually searches for the master. If manual search does not find a master,
  a fatal error occurs. Otherwise, the client updates it's masterId and eventually restarts. Restarting is
  safe because this action is reached only if "masterDo" fails
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToWithTag("b",
                                     C!INST_STATUS_LOST,
                                     "backupGetNewMaster")
        foundMaster ≜ C!FindMaster(C!INST_STATUS_ACTIVE)
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ C!RecvMsg(msg)
        ∧ IF foundMaster = C!NOT_MASTER  no live master found
          THEN ∧ exec_state' = "fatal"
               ∧ clients' = [clients EXCEPT ![msg.clientId].phase =
                                              C!PH2_COMPLETED_FATAL]
          ELSE ∧ exec_state' = exec_state
                 at this point, the live master must have been changed
               ∧ foundMaster.id ≠ clients[msg.clientId].masterId
                 change status to pending to be eligible for restart
               ∧ clients' = [clients EXCEPT ![msg.clientId].masterId =
                                              foundMaster.id,
                                            ![msg.clientId].phase =
                                              C!PH1_PENDING]
  ∧ UNCHANGED ⟨master, backup, killed⟩

C_HandlingMasterGetNewBackupFailed ≜
  The client handling the failure of the master when the client asked the master to return the new
  backup identity. The failure of the master is fatal. If a recovered master exists we should not search
  for it, because it may have the old version before masterDone.
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToWithTag("m",
```

```
                                                C!INST_STATUS_LOST,
                                                "masterGetNewBackup")
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ exec_state' = "fatal"
        ∧ clients' = [clients EXCEPT ![msg.clientId].phase =
                                                C!PH2_COMPLETED_FATAL]
        ∧ C!RecvMsg(msg)
  ∧ UNCHANGED ⟨master, backup, killed⟩

C_UpdatingBackupId ≜
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToClient("m", "newBackupId")
    IN  ∧ msg ≠ C!NOT_MESSAGE  receive new backup identity, and complete request,
                               don't restart, master is alive and up to date
        ∧ C!RecvMsg(msg)
        ∧ clients' = [clients EXCEPT ![msg.clientId].backupId = msg.backupId,
                                     ![msg.clientId].phase = C!PH2_COMPLETED]
          if all clients completed, then terminate the execution successfully
        ∧ IF ∀ c ∈ C!CLIENT_ID : clients'[c].phase = C!PH2_COMPLETED
          THEN exec_state' = "success"
          ELSE exec_state' = exec_state
  ∧ UNCHANGED ⟨master, backup, killed⟩

C_UpdatingMasterId ≜
  Client receiving a new master identify from a live backup and is preparing to restart by changing its
  phase to pending
  ∧ exec_state = "running"
  ∧ LET msg ≜ C!FindMessageToClient("b", "newMasterId")
    IN  ∧ msg ≠ C!NOT_MESSAGE
        ∧ C!RecvMsg(msg)
        ∧ clients' = [clients EXCEPT ![msg.clientId].masterId = msg.masterId,
                                     ![msg.clientId].phase = C!PH1_PENDING]
  ∧ UNCHANGED ⟨exec_state, master, backup, killed⟩

M_CreatingNewBackup ≜
  Master creating a new backup using its own exec_state. Master does not process any client requests
  during recovery
  ∧ exec_state = "running"
  ∧ LET activeM ≜ C!FindMaster(C!INST_STATUS_ACTIVE)
        activeB ≜ C!FindBackup(C!INST_STATUS_ACTIVE)
        lostB ≜ C!LastLostBackup
    IN  ∧ activeM ≠ C!NOT_MASTER  active master exists
        ∧ activeB = C!NOT_BACKUP  active backup does not exist
        ∧ lostB ≠ C!NOT_BACKUP  a lost backup exists
```

```
440         ∧ LET newBackupId ≜ lostB.id + 1
441             new backup id is the following id of the dead backup
442           IN ∧ newBackupId ≤ C!MAX_INSTANCE_ID
443              ∧ backup' = [backup EXCEPT
444                             ![newBackupId].status = C!INST_STATUS_ACTIVE,
445                             ![newBackupId].masterId = activeM.id,
446                             ![newBackupId].value = activeM.value,
447                             ![newBackupId].version = activeM.version]
448              ∧ master' = [master EXCEPT
449                             ![activeM.id].backupId = newBackupId ]
450   ∧ UNCHANGED ⟨exec_state, clients, msgs, killed⟩

452 B_CreatingNewMaster ≜
    Backup creating a new master using its own exec_state. Backup does not process any client requests during recovery
457   ∧ exec_state = "running"
458   ∧ LET activeM ≜ C!FindMaster(C!INST_STATUS_ACTIVE)
459         activeB ≜ C!FindBackup(C!INST_STATUS_ACTIVE)
460         lostM ≜ C!LastLostMaster
461     IN ∧ activeM = C!NOT_MASTER  active master does not exist
462        ∧ activeB ≠ C!NOT_BACKUP  active backup exists
463        ∧ lostM ≠ C!NOT_MASTER  a lost master exists
464        ∧ LET newMasterId ≜ lostM.id + 1
465          IN ∧ newMasterId ≤ C!MAX_INSTANCE_ID
466             ∧ master' = [master EXCEPT
467                            ![newMasterId].status = C!INST_STATUS_ACTIVE,
468                            ![newMasterId].backupId = activeB.id,
469                            ![newMasterId].value = activeB.value,
470                            ![newMasterId].version = activeB.version]
471             ∧ backup' = [backup EXCEPT
472                            ![activeB.id].masterId = newMasterId ]
473   ∧ UNCHANGED ⟨exec_state, clients, msgs, killed⟩

475 Next ≜
476   ∨ E_KillingMaster
477   ∨ E_KillingBackup
478   ∨ C_Starting
479   ∨ M_Doing
480   ∨ C_HandlingMasterDone
481   ∨ B_Doing
482   ∨ B_DetectingOldMasterId
483   ∨ C_HandlingBackupDone
484   ∨ C_HandlingMasterDoFailed
485   ∨ C_HandlingBackupDoFailed
486   ∨ M_GettingNewBackup
```

$$\lor\ B\_GettingNewMaster$$
$$\lor\ C\_HandlingBackupGetNewMasterFailed$$
$$\lor\ C\_HandlingMasterGetNewBackupFailed$$
$$\lor\ C\_UpdatingBackupId$$
$$\lor\ C\_UpdatingMasterId$$
$$\lor\ M\_CreatingNewBackup$$
$$\lor\ B\_CreatingNewMaster$$

$$Liveness \triangleq$$
$$\land \mathrm{WF}_{Vars}(E\_KillingMaster)$$
$$\land \mathrm{WF}_{Vars}(E\_KillingBackup)$$
$$\land \mathrm{WF}_{Vars}(C\_Starting)$$
$$\land \mathrm{WF}_{Vars}(M\_Doing)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingMasterDone)$$
$$\land \mathrm{WF}_{Vars}(B\_Doing)$$
$$\land \mathrm{WF}_{Vars}(B\_DetectingOldMasterId)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingBackupDone)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingMasterDoFailed)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingBackupDoFailed)$$
$$\land \mathrm{WF}_{Vars}(M\_GettingNewBackup)$$
$$\land \mathrm{WF}_{Vars}(B\_GettingNewMaster)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingBackupGetNewMasterFailed)$$
$$\land \mathrm{WF}_{Vars}(C\_HandlingMasterGetNewBackupFailed)$$
$$\land \mathrm{WF}_{Vars}(C\_UpdatingBackupId)$$
$$\land \mathrm{WF}_{Vars}(C\_UpdatingMasterId)$$
$$\land \mathrm{WF}_{Vars}(M\_CreatingNewBackup)$$
$$\land \mathrm{WF}_{Vars}(B\_CreatingNewMaster)$$

Specification

$$Spec \triangleq \ Init \land \Box[Next]_{Vars} \land Liveness$$

THEOREM $Spec \implies \Box(TypeOK \land StateOK)$

```
1  ---------- MODULE Commons ----------
2  EXTENDS Integers
3
4  CONSTANTS CLIENT_NUM,   the number of clients
5            MAX_KILL      maximum allowed kill events
6
7  VARIABLES exec_state,    the execution state of the program: running, success, or fatal
8            clients,      clients sending value update requests to master and backup
9            master,       array of master instances, only one is active
10           backup,       array of backup instances, only one is active
11           msgs,         in-flight messages
12           killed        number of invoked kill actions to master or backup
13 ------------------------------------
14
15  Identifiers related to master and backup instance ids
16 FIRST_ID ≜ 1
17 MAX_INSTANCE_ID ≜ MAX_KILL + 1
18 INSTANCE_ID ≜ FIRST_ID .. MAX_INSTANCE_ID
19 UNKNOWN_ID ≜ 0
20 NOT_INSTANCE_ID ≜ −1
21
22  Identifiers related to master and backup instance statuses
23 INST_STATUS_NULL ≜ "null"   null, not used yet
24 INST_STATUS_ACTIVE ≜ "active"   active and handling client requests
25 INST_STATUS_LOST ≜ "lost"   lost
26 NOT_STATUS ≜ "invalid"   invalid status
27 INSTANCE_STATUS ≜ {INST_STATUS_NULL,
28                   INST_STATUS_ACTIVE,
29                   INST_STATUS_LOST}
30
31  Master instance record structure
32 Master ≜ [id : INSTANCE_ID, backupId : INSTANCE_ID ∪ {UNKNOWN_ID},
33          status : INSTANCE_STATUS, value : Nat, version : Nat]
34
35  Invalid master instance
36 NOT_MASTER ≜ [id ↦ NOT_INSTANCE_ID, backupId ↦ NOT_INSTANCE_ID,
37               status ↦ NOT_STATUS, value ↦ −1, version ↦ −1]
38
39  Backup instance record structure
40 Backup ≜ [id : INSTANCE_ID, masterId : INSTANCE_ID ∪ {UNKNOWN_ID},
41          status : INSTANCE_STATUS, value : Nat, version : Nat]
42
43  Invalid backup instance
44 NOT_BACKUP ≜ [id ↦ NOT_INSTANCE_ID, masterId ↦ NOT_INSTANCE_ID,
45               status ↦ NOT_STATUS, value ↦ −1, version ↦ −1]
46
47 LastLostMaster ≜
    Return the lost master, or NOT_MASTER if master is alive
```

```
51    LET mset ≜ {m ∈ INSTANCE_ID : master[m].status = INST_STATUS_LOST}
52    IN  IF mset = {} THEN NOT_MASTER
53        ELSE  master[(CHOOSE n ∈ mset : ∀ m ∈ mset : n ≥ m)]

55  FindMaster(mStatus) ≜
      Return the master with given status or NOT_MASTER otherwise
59    LET mset ≜ {m ∈ INSTANCE_ID : master[m].status = mStatus}
60    IN  IF mset = {} THEN NOT_MASTER
61        ELSE  master[(CHOOSE x ∈ mset : TRUE)]

63  LastKnownMaster ≜
      Return the last known master, whether active or lost
67    LET mset ≜ {m ∈ INSTANCE_ID : master[m].status ≠ INST_STATUS_NULL}
68    IN  master[(CHOOSE n ∈ mset : ∀ m ∈ mset : n ≥ m)]

70  FindBackup(bStatus) ≜
      Return the backup with given status or NOT_BACKUP otherwise
74    LET bset ≜ {b ∈ INSTANCE_ID : backup[b].status = bStatus}
75    IN  IF bset = {} THEN NOT_BACKUP
76        ELSE  backup[(CHOOSE x ∈ bset : TRUE)]

78  LastLostBackup ≜
      Return the lost backup, or NOT_BACKUP if backup is alive
82    LET bset ≜ {b ∈ INSTANCE_ID : backup[b].status = INST_STATUS_LOST}
83    IN  IF bset = {} THEN NOT_BACKUP
84        ELSE  backup[(CHOOSE n ∈ bset : ∀ m ∈ bset : n ≥ m)]

86  LastKnownBackup ≜
      Return the last known backup, whether active or lost
90    LET bset ≜ {b ∈ INSTANCE_ID : backup[b].status ≠ INST_STATUS_NULL}
91    IN  backup[(CHOOSE n ∈ bset : ∀ m ∈ bset : n ≥ m)]

93  ----------------------------------------------------------------

94    Identifiers related to client ids and phases
95  CLIENT_ID ≜ 1 .. CLIENT_NUM
96  NOT_CLIENT_ID ≜ −1

98    client phases
99  CLIENT_PHASE ≜ 1 .. 4
100 PH1_PENDING ≜ 1
101 PH2_WORKING ≜ 2
102 PH2_COMPLETED ≜ 3
103 PH2_COMPLETED_FATAL ≜ 4
104 NOT_CLIENT_PHASE ≜ −1

106   Client record structure
107 Client ≜ [id : CLIENT_ID, phase : CLIENT_PHASE, value : Nat,
```

```
108             the master instance last communicated with
109           masterId : INSTANCE_ID,
110             the backup instance last communicated with, initially unknown
111           backupId : INSTANCE_ID ∪ {UNKNOWN_ID}
112         ]

114  Invalid client instance
115 NOT_CLIENT ≜ [id ↦ NOT_CLIENT_ID, phase ↦ NOT_CLIENT_PHASE, value ↦ 0]

117 FindClient(phase) ≜
      Return a client matching the given phase, or NOT_CLIENT otherwise
121   LET cset ≜ {c ∈ CLIENT_ID : clients[c].phase = phase}
122   IN  IF cset = {} THEN NOT_CLIENT
123       ELSE clients[(CHOOSE x ∈ cset : TRUE)]

125 ----------------------------------------------------------------

126  Message record structure
127 Messages ≜ [from : {"c", "m", "b", "sys"}, to : {"c", "m", "b"},
128               clientId  : CLIENT_ID,
129               masterId : INSTANCE_ID ∪ {UNKNOWN_ID},
130               backupId : INSTANCE_ID ∪ {UNKNOWN_ID},
131               value : Nat,
132               tag : {"masterDo", "masterDone",
133                      "backupDo", "backupDone",
134                      "masterGetNewBackup", "newBackupId",
135                      "backupGetNewMaster", "newMasterId"
136                     }]

138  Invalid message instance
139 NOT_MESSAGE ≜ [from ↦ "na", to ↦ "na"]

141 SendMsg(m) ≜
      Add message to the msgs set
145   msgs' = msgs ∪ {m}

147 RecvMsg(m) ≜
      Delete message from the msgs set
151   msgs' = msgs \ {m}

153 ReplaceMsg(toRemove, toAdd) ≜
      Remove an existing message and add another one
157   msgs' = (msgs \ {toRemove}) ∪ {toAdd}

159 FindMessageToWithTag(to, status, optionalTag) ≜
      Return a message matching the given criteria, or NOT_MESSAGE otherwise
163   LET mset ≜ {m ∈ msgs : ∧ m.to = to
164                           ∧ IF to = "m"
```

```
165                                 THEN master[m.masterId].status = status
166                                 ELSE IF to = "b"
167                                 THEN backup[m.backupId].status = status
168                                 ELSE FALSE
169                               ∧ IF optionalTag = "NA"
170                                 THEN TRUE
171                                 ELSE m.tag = optionalTag}
172    IN IF mset = {} THEN NOT_MESSAGE
173       ELSE (CHOOSE x ∈ mset : TRUE)

175 FindMessageTo(to, status) ≜ FindMessageToWithTag(to, status, "NA")

177 FindMessageToClient(from, tag) ≜
      Return a message sent to client matching given criteria, or NOT_MESSAGE otherwise
182    LET mset ≜ {m ∈ msgs : ∧ m.from = from
183                           ∧ m.to = "c"
184                           ∧ m.tag = tag}
185    IN IF mset = {} THEN NOT_MESSAGE
186       ELSE (CHOOSE x ∈ mset : TRUE)

188
```

List of Abbreviations

2PC	Two-Phase Commit
ABFT	Algorithmic-Based Fault Tolerance
AMPI	Adaptive MPI
APGAS	Asynchronous Partitioned Global Address Space
API	Application Programming Interface
ARMCI	Aggregate Remote Memory Copy Interface
BLCR	Berkeley Lab Checkpoint/Restart
CC	concurrency control
DMTCP	Distributed MultiThreaded CheckPointing
DTM	Distributed Transactional Memory
EA	Early Acquire
FMI	Fault Tolerant Messaging Interface
FTWG	Fault Tolerance Working Group
GASNet	Global Address Space Networking
GASPI	Global Address Space Programming Interface
GC	Garbage Collection
GML	Global Matrix Library
HBI	Happens-Before Invariance
HPC	High Performance Computing
HPCS	High Productivity Computing Systems
HTM	hardware transactional memory

LA Late Acquire

MPI Message Passing Interface

MPI-ULFM MPI User Level Failure Mitigation

MTBF Mean Time Between Failures

O-dist optimistic distributed finish

O-p0 optimistic place-zero finish

OCR Open Community Runtime

P-dist pessimistic distributed finish

P-p0 pessimistic place-zero finish

PAMI the Parallel Active Message Interface

PDE Partial Differential Equation

PFS Parallel File System

PGAS Partitioned Global Address Space

RAS Reliability, Availability, and Serviceability

RDD Resilient Distributed Datasets

RDMA Remote Direct Memory Access

RL Read Locking

RTS Run Through Stabilization

RV Read Versioning

RX10 Resilient X10

SPMD Single Program Multiple Data

SSCA Scalable Synthetic Compact Application

SSH Secure Shell

TD termination detection

TLA Temporal Logic of Actions

TM Transactional Memory

uGNI user Generic Network Interface

UL Undo-Logging

UPC Unified Parallel C

WB Write-Buffering

X10RT X10 Runtime Transport

Bibliography

Acun, B.; Gupta, A.; Jain, N.; Langer, A.; Menon, H.; Mikida, E.; Ni, X.; Robson, M.; Sun, Y.; Totoni, E.; et al., 2014. Parallel programming with migratable objects: Charm++ in practice. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 647–658. IEEE Press. (cited on page 28)

Aiken, A.; Bauer, M.; and Treichler, S., 2014. Realm: An event-based low-level runtime for distributed memory architectures. In *Proc. 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 263–275. IEEE. (cited on pages 17, 30, and 31)

Akka, 2018. Akka Documentation. https://akka.io/docs/. (cited on page 29)

Ali, M. M.; Southern, J.; Strazdins, P.; and Harding, B., 2014. Application level fault recovery: Using fault-tolerant Open MPI in a PDE solver. In *Proc. Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, 1169–1178. IEEE. (cited on pages 43 and 51)

Ali, M. M.; Strazdins, P. E.; Harding, B.; Hegland, M.; and Larson, J. W., 2015. A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In *Proc. International Conference on High Performance Computing & Simulation*, HPCS, 499–507. (cited on pages 43 and 68)

Ali, N.; Krishnamoorthy, S.; Govind, N.; and Palmer, B., 2011a. A redundant communication approach to scalable fault tolerance in PGAS programming models. In *Proc. 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 24–31. IEEE. (cited on page 25)

Ali, N.; Krishnamoorthy, S.; Halappanavar, M.; and Daily, J., 2011b. Tolerating correlated failures for generalized Cartesian distributions via bipartite matching. In *Proc. 8th ACM International Conference on Computing Frontiers*. ACM. (cited on page 25)

Almeida, R.; Neto, A. A.; and Vieira, M., 2013. Score: An across-the-board metric for computer systems resilience benchmarking. In *Proc. 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, 1–8. IEEE. (cited on page 9)

Ansel, J.; Arya, K.; and Cooperman, G., 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. International Symposium on Parallel & Distributed Processing (IPDPS'09).*, 1–12. IEEE. (cited on pages 25 and 162)

Augonnet, C.; Thibault, S.; Namyst, R.; and Wacrenier, P.-A., 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23, 2 (2011), 187–198. (cited on page 31)

Bader, D. A. and Madduri, K., 2005. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. Technical report, Georgia Institute of Technology. (cited on page 148)

Bartsch, V.; Machado, R.; Merten, D.; Rahn, M.; and Pfreundt, F.-J., 2017. GASPI/GPI In-memory Checkpointing Library. In *European Conference on Parallel Processing*, 497–508. Springer. (cited on page 24)

Bauer, M.; Treichler, S.; Slaughter, E.; and Aiken, A., 2012. Legion: expressing locality and independence with logical regions. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, 1–11. IEEE. (cited on page 30)

Bauer, M. E., 2014. *Legion: programming distributed heterogeneous architectures with logical regions*. Ph.D. thesis, Stanford University. (cited on page 30)

Bergman, K.; Borkar, S.; Campbell, D.; Carlson, W.; Dally, W.; Denneau, M.; Franzon, P.; Harrod, W.; Hill, K.; Hiller, J.; et al., 2008. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15 (2008). (cited on page 1)

Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; and Thelin, J., 2014. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014–41, Microsoft Research. https://pdfs.semanticscholar.org/aacf/bf0d34bc24dc3b72e56719ec083759a072ce.pdf. (cited on page 29)

Bernstein, P. A. and Goodman, N., 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13, 2 (1981), 185–221. (cited on page 111)

Bernstein, P. A. and Goodman, N., 1984. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems (TODS)*, 9, 4 (1984), 596–615. (cited on page 16)

Bernstein, P. A.; Hadzilacos, V.; and Goodman, N., 1987. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA. (cited on page 119)

Bland, W.; Bosilca, G.; Bouteiller, A.; Herault, T.; and Dongarra, J., 2012a. A Proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical Report ut-cs-12-693, University of Tennessee Electrical Engineering and Computer Science. (cited on page 42)

Bland, W.; Bouteiller, A.; Herault, T.; Bosilca, G.; and Dongarra, J., 2013. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27, 3 (2013), 244–254. (cited on pages 10, 42, and 49)

Bland, W.; Bouteiller, A.; Herault, T.; Hursey, J.; Bosilca, G.; and Dongarra, J. J., 2012b. An Evaluation of User-level Failure Mitigation Support in MPI. In *Proc. EuroMPI'12* (Vienna, Austria, 2012), 193–203. doi:10.1007/978-3-642-33518-1_24. (cited on pages 1, 22, and 42)

Blumofe, R. D.; Joerg, C. F.; Kuszmaul, B. C.; Leiserson, C. E.; Randall, K. H.; and Zhou, Y., 1995. *Cilk: An efficient multithreaded runtime system*, vol. 30. ACM. (cited on page 70)

Blumofe, R. D. and Leiserson, C. E., 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46, 5 (1999), 720–748. (cited on page 71)

Bocchino, R. L.; Adve, V. S.; and Chamberlain, B. L., 2008. Software transactional memory for large scale clusters. In *Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 247–258. ACM. (cited on pages 118, 120, 144, 145, and 148)

Bosilca, G.; Bouteiller, A.; Cappello, F.; Djilali, S.; Fedak, G.; Germain, C.; Herault, T.; Lemarinier, P.; Lodygensky, O.; Magniette, F.; et al., 2002. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. ACM/IEEE Conference on Supercomputing*, 29–29. IEEE. (cited on page 20)

Bosilca, G.; Bouteiller, A.; Guermouche, A.; Herault, T.; Robert, Y.; Sens, P.; and Dongarra, J., 2016. Failure Detection and Propagation in HPC systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, 312–322. IEEE. (cited on page 22)

Bougeret, M.; Casanova, H.; Robert, Y.; Vivien, F.; and Zaidouni, D., 2014. Using group replication for resilience on exascale systems. *The International Journal of High Performance Computing Applications*, 28, 2 (2014), 210–224. (cited on page 16)

Bouteiller, A.; Bosilca, G.; and Venkata, M. G., 2016. Surviving errors with openshmem. In *Workshop on OpenSHMEM and Related Technologies*, 66–81. Springer. (cited on page 25)

Bungart, M. and Fohry, C., 2017a. A Malleable and Fault-Tolerant Task Pool Framework for X10. In *Proc. International Conference on Cluster Computing (CLUSTER)*, 749–757. IEEE. (cited on page 57)

Bungart, M. and Fohry, C., 2017b. Extending the MPI backend of X10 by elasticity. In *EuroMPI Poster*. (cited on page 37)

Bykov, S.; Geller, A.; Kliot, G.; Larus, J. R.; Pandya, R.; and Thelin, J., 2011. Orleans: cloud computing for everyone. In *Proc. 2nd ACM Symposium on Cloud Computing*, 16. ACM. (cited on page 29)

Cao, C.; Herault, T.; Bosilca, G.; and Dongarra, J., 2015. Design for a soft error resilient dynamic task-based runtime. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 765–774. IEEE. (cited on pages 12 and 31)

Cappello, F., 2009. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 23, 3 (2009), 212–226. (cited on pages 10 and 107)

Cappello, F.; Geist, A.; Gropp, B.; Kale, L.; Kramer, B.; and Snir, M., 2009. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23, 4 (2009), 374–388. (cited on pages 1 and 16)

Chakrabarti, D.; Zhan, Y.; and Faloutsos, C., 2004. R-MAT: A recursive model for graph mining. In *Proc. SIAM International Conference on Data Mining*, 442–446. SIAM. (cited on page 148)

Chakravorty, S. and Kale, L. V., 2004. A fault tolerant protocol for massively parallel systems. In *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 212. IEEE. doi:10.1109/IPDPS.2004.1303244. (cited on page 28)

Chakravorty, S.; Mendes, C. L.; and Kalé, L. V., 2006. Proactive fault tolerance in MPI applications via task migration. In *International Conference on High-Performance Computing*, 485–496. Springer. (cited on page 29)

Chamberlain, B. L.; Callahan, D.; and Zima, H. P., 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 21, 3 (2007), 291–312. (cited on pages 1, 2, 26, and 70)

Charles, P.; Grothoff, C.; Saraswat, V.; Donawa, C.; Kielstra, A.; Ebcioglu, K.; Von Praun, C.; and Sarkar, V., 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, 519–538. ACM. (cited on pages 1, 2, and 27)

Chen, Y.; Wei, X.; Shi, J.; Chen, R.; and Chen, H., 2016. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th European Conference on Computer Systems*, 26. ACM. (cited on pages 33 and 169)

Chen, Z. and Dongarra, J., 2008. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19, 12 (2008), 1628–1641. (cited on page 17)

Chien, A.; Balaji, P.; Beckman, P.; Dun, N.; Fang, A.; Fujita, H.; Iskra, K.; Rubenstein, Z.; Zheng, Z.; Schreiber, R.; et al., 2015. Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience. *Procedia Computer Science*, 51 (2015), 29–38. (cited on page 25)

Coarfa, C.; Dotsenko, Y.; Mellor-Crummey, J.; Cantonnet, F.; El-Ghazawi, T.; Mohanti, A.; Yao, Y.; and Chavarría-Miranda, D., 2005. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proc. 10th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 36–47. ACM. (cited on page 23)

Coti, C.; Herault, T.; Lemarinier, P.; Pilard, L.; Rezmerita, A.; Rodriguez, E.; and Cappello, F., 2006. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Proc. 2006 ACM/IEEE conference on Supercomputing*, 127. ACM. (cited on page 20)

Crafa, S.; Cunningham, D.; Saraswat, V.; Shinnar, A.; and Tardieu, O., 2014. Semantics of (resilient) X10. In *European Conference on Object-Oriented Programming*, 670–696. Springer. (cited on page 2)

Cunningham, D.; Grove, D.; Herta, B.; Iyengar, A.; Kawachiya, K.; Murata, H.; Saraswat, V.; Takeuchi, M.; and Tardieu, O., 2014. Resilient X10: Efficient Failure-Aware Programming. In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)* (Orlando, Florida, USA, 2014), 67–80. ACM. (cited on pages 2, 5, 27, 33, 36, 37, 70, 73, 76, 89, 90, 91, 96, and 106)

Daily, J.; Vishnu, A.; van Dam, H.; Palmer, B.; and Kerbyson, D. J., 2014. On the Suitability of MPI as a PGAS Runtime. In *International Conference on High Performance Computing (HiPC'14)*. (cited on page 23)

Davies, T.; Karlsson, C.; Liu, H.; Ding, C.; and Chen, Z., 2011. High performance LINPACK benchmark: a fault tolerant implementation without checkpointing. In *Proc. International Conference on Supercomputing*, 162–171. ACM. (cited on page 17)

Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51, 1 (2008), 107–113. (cited on page 17)

Dijkstra, E. W. and Scholten, C. S., 1980. Termination detection for diffusing computations. *Information Processing Letters*, 11, 1 (1980), 1–4. (cited on pages 70 and 72)

Dongarra, J.; Beckman, P.; Moore, T.; Aerts, P.; Aloisio, G.; Andre, J.-C.; Barkai, D.; Berthou, J.-Y.; Boku, T.; Braunschweig, B.; et al., 2011. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25, 1 (2011), 3–60. (cited on pages 1 and 16)

Dragojević, A.; Narayanan, D.; Nightingale, E. B.; Renzelmann, M.; Shamis, A.; Badam, A.; and Castro, M., 2015. No compromises: distributed transactions with

consistency, availability, and performance. In *Proc. 25th Symposium on Operating Systems Principles*, 54–70. ACM. (cited on pages 33 and 169)

Du, P.; Bouteiller, A.; Bosilca, G.; Herault, T.; and Dongarra, J., 2012. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47, 8 (2012), 225–234. (cited on pages 12 and 17)

Du, P.; Luszczek, P.; and Dongarra, J., 2011. High performance dense linear system solver with soft error resilience. In *Proc. International Conference on Cluster Computing*, 272–280. IEEE. (cited on page 12)

Egwutuoha, I. P.; Levy, D.; Selic, B.; and Chen, S., 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65, 3 (2013), 1302–1326. (cited on pages 10 and 16)

El-Ghazawi, T. and Smith, L., 2006. UPC: Unified Parallel C. In *Proc. ACM/IEEE conference on Supercomputing*, 27. ACM. (cited on page 23)

Elnozahy, E. N.; Alvisi, L.; Wang, Y.-M.; and Johnson, D. B., 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34, 3 (2002), 375–408. (cited on pages 10 and 15)

Fagg, G. E. and Dongarra, J. J., 2000. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 346–353. Springer. (cited on pages 1 and 21)

Fanfarillo, A.; Garain, S. K.; Balsara, D.; and Nagle, D., 2019. Resilient computational applications using Coarray Fortran. *Parallel Computing*, 81 (2019), 58–67. (cited on pages 24 and 67)

Fang, A.; Laguna, I.; Sato, K.; Islam, T.; and Mohror, K., 2016. Fault Tolerance Assistant (FTA): An Exception Handling Programming Model for MPI Applications. Technical report, Lawrence Livermore National Laboratory. doi:10.2172/1258538. https://e-reports-ext.llnl.gov/pdf/820672.pdf. (cited on page 2)

Feng, S.; Gupta, S.; Ansari, A.; and Mahlke, S., 2010. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, vol. 38, 385–396. ACM. (cited on page 12)

Ferreira, K.; Stearley, J.; Laros, J. H.; Oldfield, R.; Pedretti, K.; Brightwell, R.; Riesen, R.; Bridges, P. G.; and Arnold, D., 2011. Evaluating the viability of process replication reliability for exascale systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 1–12. IEEE. (cited on pages 1 and 21)

Fiala, D.; Mueller, F.; Engelmann, C.; Riesen, R.; Ferreira, K.; and Brightwell, R., 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 78. IEEE Computer Society Press. (cited on page 21)

Fohry, C.; Bungart, M.; and Plock, P., 2017. Fault tolerance for lifeline-based global load balancing. *Journal of Software Engineering and Applications*, 10, 13 (2017), 925–958. (cited on page 18)

Gainaru, A.; Cappello, F.; Snir, M.; and Kramer, W., 2012. Fault prediction under the microscope: A closer look into HPC systems. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 77. IEEE Computer Society Press. (cited on page 14)

Gamell, M.; Katz, D. S.; Teranishi, K.; Heroux, M. A.; Van der Wijngaart, R. F.; Mattson, T. G.; and Parashar, M., 2016. Evaluating online global recovery with Fenix using application-aware in-memory checkpointing techniques. In *Proc. 45th International Conference on Parallel Processing Workshops (ICPPW)*, 346–355. IEEE. (cited on page 67)

Garcia-Molina, H., 1982. Elections in a distributed computing system. *IEEE transactions on Computers*, , 1 (1982), 48–59. (cited on page 112)

Garg, R.; Vienne, J.; and Cooperman, G., 2016. System-level transparent checkpointing for OpenSHMEM. In *Workshop on OpenSHMEM and Related Technologies*, 52–65. Springer. (cited on page 25)

GASNet, 2018. Gasnet. https://gasnet.lbl.gov. (cited on page 23)

Graham, R.; Hursey, J.; Vallée, G.; Naughton, T.; and Boehm, S., 2012. The Impact of a Fault Tolerant MPI on Scalable Systems Services and Applications. In *Proc. Cray Users Group Conference*. (cited on page 17)

Gropp, W. and Lusk, E., 2004. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18, 3 (2004), 363–372. (cited on page 22)

Grove, D.; Hamouda, S. S.; Herta, B.; Iyengar, A.; Kawachiya, K.; Milthorpe, J.; Saraswat, V.; Shinnar, A.; Takeuchi, M.; Tardieu, O.; et al., 2019. Failure Recovery in Resilient X10 (accepted). *ACM Transactions on Programming Languages and Systems*, (2019). (cited on pages 7, 93, and 106)

Guo, Y.; Barik, R.; Raman, R.; and Sarkar, V., 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 1–12. IEEE. (cited on pages 2 and 71)

Hadoop/ZooKeeper. Hadoop/zookeeper. https://wiki.apache.org/hadoop/ZooKeeper. (cited on page 89)

Hakkarinen, D. and Chen, Z., 2010. Algorithmic Cholesky factorization fault recovery. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, 1–10. IEEE. (cited on page 17)

Hamouda, S. S.; Herta, B.; Milthorpe, J.; Grove, D.; and Tardieu, O., 2016. Resilient X10 over MPI user level failure mitigation. In *Proc. 6th ACM SIGPLAN Workshop on X10*, 18–23. ACM. (cited on pages 7, 41, 45, 46, 105, and 106)

Hamouda, S. S. and Milthorpe, J., 2019. Resilient Optimistic Termination Detection for the Async-Finish Model (to appear). In *ISC High Performance, Frankfurt, Germany*. (cited on pages 7 and 69)

Hamouda, S. S.; Milthorpe, J.; Strazdins, P. E.; and Saraswat, V., 2015. A resilient framework for iterative linear algebra applications in X10. In *Proc. International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, 970–979. IEEE. (cited on pages 5, 7, 105, 106, 108, 110, and 152)

Hao, P.; Shamis, P.; Venkata, M. G.; Pophale, S.; Welch, A.; Poole, S.; and Chapman, B., 2014a. Fault tolerance for OPENSHMEM. In *Proc. 8th International Conference on Partitioned Global Address Space Programming Models*, 23. ACM. (cited on page 25)

Hao, Z.; Xie, C.; Chen, H.; and Zang, B., 2014b. X10-FT: transparent fault tolerance for APGAS language and runtime. *Parallel Computing*, 40, 2 (2014), 136–156. (cited on page 27)

Harding, B. and Hegland, M., 2013. A parallel fault tolerant combination technique. In *Proc. International Conference on Parallel Computing, (ParCo'13)*, 584–592. (cited on page 68)

Harding, R.; Van Aken, D.; Pavlo, A.; and Stonebraker, M., 2017. An evaluation of distributed concurrency control. *Proc. VLDB Endowment*, 10, 5 (2017), 553–564. (cited on page 16)

Harris, T.; Larus, J.; and Rajwar, R., 2010. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5, 1 (2010), 1–263. (cited on page 112)

Hassani, A.; Skjellum, A.; Bangalore, P. V.; and Brightwell, R., 2015. Practical resilient cases for FA-MPI, a transactional fault-tolerant MPI. In *Proc. 3rd Workshop on Exascale MPI*, 1. ACM. (cited on page 22)

Hayashi, A.; Paul, S. R.; Grossman, M.; Shirako, J.; and Sarkar, V., 2017. Chapel-on-X: Exploring Tasking Runtimes for PGAS Languages. In *Proc. 3rd International Workshop on Extreme Scale Programming Models and Middleware*, 5. ACM. (cited on pages 1 and 26)

Hazelcast, Inc., 2014. Hazelcast 3.4. https://hazelcast.com/. (cited on page 106)

Herault, T.; Bouteiller, A.; Bosilca, G.; Gamell, M.; Teranishi, K.; Parashar, M.; and Dongarra, J., 2015. Practical scalable consensus for pseudo-synchronous distributed systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, 1–12. IEEE. (cited on page 56)

Herlihy, M. and Moss, J. E. B., 1993. *Transactional Memory: Architectural support for lock-free data structures*, vol. 21. ACM. (cited on page 111)

Huang, C.; Lawlor, O.; and Kale, L. V., 2003. Adaptive MPI. In *International workshop on languages and compilers for parallel computing*, 306–322. Springer. (cited on page 28)

Huang, K.-H. and Abraham, J. A., 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100, 6 (1984), 518–528. (cited on page 17)

Hursey, J.; Graham, R. L.; Bronevetsky, G.; Buntinas, D.; Pritchard, H.; and Solt, D. G., 2011. Run-Through Stabilization: An MPI proposal for process fault tolerance. In *European MPI Users' Group Meeting*, 329–332. Springer. (cited on pages 1 and 22)

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A.; and Fetterly, D., 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, vol. 41, 59–72. ACM. (cited on page 17)

Jain, N.; Bhatele, A.; Yeom, J.-S.; Adams, M. F.; Miniati, F.; Mei, C.; and Kale, L. V., 2015. Charm++ and MPI: Combining the best of both worlds. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 655–664. IEEE. (cited on page 28)

Kaiser, H.; Heller, T.; Adelstein-Lelbach, B.; Serio, A.; and Fey, D., 2014. HPX: A task based programming model in a global address space. In *Proc. 8th International Conference on Partitioned Global Address Space Programming Models*, 6. ACM. (cited on page 31)

Kale, L. V. and Krishnan, S., 1993. CHARM++: a portable concurrent object oriented system based on C++. In *Proc. 8th annual conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, 91–108. ACM. (cited on page 28)

Karlin, I.; Keasler, J.; and Neely, R., 2013. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973. (cited on page 158)

Keidar, I. and Dolev, D., 1998. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences*, 57, 3 (1998), 309–324. (cited on pages 112 and 119)

Kurt, M. C.; Krishnamoorthy, S.; Agrawal, K.; and Agrawal, G., 2014. Fault-tolerant dynamic task graph scheduling. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 719–730. IEEE Press. (cited on page 31)

Laguna, I.; Richards, D. F.; Gamblin, T.; Schulz, M.; and de Supinski, B. R., 2014. Evaluating user-level fault tolerance for MPI applications. In *Proc. 21st European MPI Users' Group Meeting*, 57. ACM. (cited on pages 18 and 43)

Laguna, I.; Richards, D. F.; Gamblin, T.; Schulz, M.; de Supinski, B. R.; Mohror, K.; and Pritchard, H., 2016. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications*, 30, 3 (2016), 305–319. (cited on pages 2 and 67)

Lai, T.-H. and Wu, L.-F., 1995. An (n-1)-resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1 (1995), 63–78. (cited on pages 70 and 72)

Lemarinier, P.; Bouteiller, A.; Herault, T.; Krawezik, G.; and Cappello, F., 2004. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *Proc. International Conference on Cluster Computing*, 115–124. IEEE. (cited on page 20)

Liang, Y.; Zhang, Y.; Sivasubramaniam, A.; Jette, M.; and Sahoo, R., 2006. Bluegene/L failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN)*, 425–434. IEEE. (cited on page 14)

Lifflander, J.; Krishnamoorthy, S.; and Kale, L. V., 2012. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proc. 21st International Symposium on High-Performance Parallel and Distributed Computing*, 137–148. ACM. (cited on page 18)

Lifflander, J.; Miller, P.; and Kale, L., 2013. Adoption protocols for fanout-optimal fault-tolerant termination detection. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'13*. ACM. (cited on pages 70, 72, 73, 83, and 107)

Losada, N.; Cores, I.; Martín, M. J.; and González, P., 2017. Resilient MPI applications using an application-level checkpointing framework and ULFM. *The Journal of Supercomputing*, 73, 1 (2017), 100–113. (cited on page 68)

Maloney, A. and Goscinski, A., 2009. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*, 21, 12 (2009), 1632–1666. (cited on pages 10 and 15)

Matocha, J. and Camp, T., 1998. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43, 3 (1998), 207–221. (cited on page 72)

Mattson, T. G.; Cledat, R.; Cavé, V.; Sarkar, V.; Budimlić, Z.; Chatterjee, S.; Fryman, J.; Ganev, I.; Knauerhase, R.; Lee, M.; et al., 2016. The Open Community Runtime: A Runtime System for Extreme Scale Computing. In *Proc. High Performance Extreme Computing Conference (HPEC '16)*, 1–7. IEEE. (cited on pages 12, 17, and 30)

Meneses, E.; Ni, X.; and Kalé, L. V., 2011. Design and analysis of a message logging protocol for fault tolerant multicore systems. Technical report, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign. http://charm.cs.uiuc.edu/newPapers/11-30/paper.pdf. (cited on page 28)

Meneses, E.; Ni, X.; and Kalé, L. V., 2012. A message-logging protocol for multicore systems. In *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 1–6. IEEE. (cited on pages 72, 83, and 107)

Meneses, E.; Ni, X.; Zheng, G.; Mendes, C. L.; and Kale, L. V., 2014. Using Migratable Objects to Enhance Fault Tolerance Schemes in Supercomputers. *IEEE transactions on parallel and distributed systems*, 26, 7 (2014), 2061–2074. (cited on pages 16 and 17)

Min, S.-J.; Iancu, C.; and Yelick, K., 2011. Hierarchical work stealing on manycore clusters. In *5th Conference on Partitioned Global Address Space Programming Models (PGAS11)*, vol. 625. (cited on page 24)

Mohan, C.; Lindsay, B.; and Obermarck, R., 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11, 4 (1986), 378–396. (cited on pages 112, 115, 119, and 121)

Moody, A.; Bronevetsky, G.; Mohror, K.; and Supinski, B. R. d., 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–11. IEEE Computer Society. (cited on pages 72, 83, and 107)

Moss, J. E. B., 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press. (cited on page 117)

Moss, J. E. B. and Hosking, A. L., 2006. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63, 2 (2006), 186–201. (cited on page 113)

MPI-4, 2018. MPI 4.0. http://mpi-forum.org/mpi-40/. (cited on page 42)

MPI-ULFM, 2018. MPI-ULFM Specification. http://fault-tolerance.org/ulfm/ulfm-specification. (cited on page 48)

MPICH, 2018. High-Performance Portable MPI. https://www.mpich.org. (cited on page 19)

Mukherjee, S. S.; Emer, J.; and Reinhardt, S. K., 2005. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, 243–247. IEEE. (cited on page 12)

Ni, X.; Meneses, E.; Jain, N.; and Kalé, L. V., 2013. ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM. doi:10.1145/2503210.2503266. (cited on pages 15 and 28)

Ni, X.; Meneses, E.; and Kalé, L. V., 2012. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 364–372. IEEE. (cited on page 28)

Nieplocha, J.; Harrison, R. J.; and Littlefield, R. J., 1996. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10, 2 (1996), 169–189. (cited on page 25)

Nieplocha, J.; Tipparaju, V.; Krishnan, M.; and Panda, D. K., 2006. High performance remote memory access communication: The ARMCI approach. *The International Journal of High Performance Computing Applications*, 20, 2 (2006), 233–253. (cited on page 23)

Numrich, R. W., 2018. *Parallel Programming with Co-arrays: Parallel Programming in Fortran*. Chapman and Hall/CRC. (cited on page 24)

Numrich, R. W. and Reid, J., 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, vol. 17, 1–31. ACM. (cited on page 24)

OpenCoarrays, 2019. OpenCoarrays. http://www.opencoarrays.org. (cited on pages 24 and 67)

OpenMPI, 2018. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org. (cited on pages 19 and 43)

OpenSHMEM, 2017. OpenSHMEM 1.4. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf. (cited on page 25)

Panagiotopoulou, K. and Loidl, H.-W., 2015. Towards Resilient Chapel: Design and implementation of a transparent resilience mechanism for Chapel. In *Proc. 3rd International Conference on Exascale Applications and Software*, 86–91. University of Edinburgh. (cited on page 26)

Pauli, S.; Kohler, M.; and Arbenz, P., 2013. A fault tolerant implementation of Multi-Level Monte Carlo methods. In *Parallel Computing: Accelerating Computational Science and Engineering (PARCO)*, vol. 13, 471–480. (cited on page 43)

Rizzi, F.; Morris, K.; Sargsyan, K.; Mycek, P.; Safta, C.; Debusschere, B.; LeMaitre, O.; and Knio, O., 2016. ULFM-MPI implementation of a resilient task-based partial differential equations preconditioner. In *Proc. ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, 19–26. ACM. (cited on pages 14 and 68)

Rodríguez, G.; Martín, M. J.; González, P.; Tourino, J.; and Doallo, R., 2010. CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience*, 22, 6 (2010), 749–766. (cited on page 68)

Ropars, T.; Lefray, A.; Kim, D.; and Schiper, A., 2015. Efficient Process Replication for MPI Applications: Sharing Work Between Replicas. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 645–654. IEEE. (cited on pages 16 and 21)

Rosenkrantz, D. J.; Stearns, R. E.; and Lewis II, P. M., 1978. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3, 2 (1978), 178–198. (cited on page 117)

Saad, M. M. and Ravindran, B., 2011. HyFlow: A high performance distributed software transactional memory framework. In *Proc. 20th International Symposium on High Performance Distributed Computing*, 265–266. ACM. (cited on page 112)

Sahoo, R. K.; Bae, M.; Vilalta, R.; Moreira, J.; Ma, S.; and Gupta, M., 2002. Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems. In *Workshop on Self-Healing, Adaptive, and Self-Managed Systems*. (cited on page 14)

Saraswat, V.; Almasi, G.; Bikshandi, G.; Cascaval, C.; Cunningham, D.; Grove, D.; Kodali, S.; Peshansky, I.; and Tardieu, O., 2010. The asynchronous partitioned global address space model. In *The First Workshop on Advances in Message Passing*, 1–8. (cited on page 1)

Saraswat, V. A.; Kambadur, P.; Kodali, S.; Grove, D.; and Krishnamoorthy, S., 2011. Lifeline-based global load balancing. In *ACM SIGPLAN Notices*, vol. 46, 201–212. ACM. (cited on page 18)

Sato, K.; Maruyama, N.; Mohror, K.; Moody, A.; Gamblin, T.; de Supinski, B. R.; and Matsuoka, S., 2012. Design and modeling of a non-blocking checkpointing system. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, 1–10. IEEE. (cited on pages 83 and 107)

Sato, K.; Moody, A.; Mohror, K.; Gamblin, T.; de Supinski, B. R.; Maruyama, N.; and Matsuoka, S., 2014. FMI: Fault tolerant messaging interface for fast and transparent recovery. In *Proc. 28th International Parallel and Distributed Processing Symposium*, 1225–1234. IEEE. (cited on pages 20 and 107)

Schroeder, B. and Gibson, G. A., 2007. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, vol. 78, 012022. IOP Publishing. (cited on page 1)

Shalf, J.; Dosanjh, S.; and Morrison, J., 2010. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, 1–25. Springer. (cited on page 1)

Shet, A.; Tipparaju, V.; and Harrison, R., 2009. Asynchronous programming in UPC: A case study and potential for improvement. In *Workshop on asynchrony in the PGAS programming model collocated with ICS*, vol. 2009. Citeseer. (cited on page 24)

Simmendinger, C.; Rahn, M.; and Gruenewald, D., 2015. The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In *Sustained Simulation Performance 2014*, 17–32. Springer. (cited on pages 24 and 169)

Skeen, D., 1981. Nonblocking commit protocols. In *Proc. 1981 ACM SIGMOD International Conference on Management of Data*, 133–142. ACM. (cited on page 16)

Squyres, J., 2011. What is an MPI eager limit? https://blogs.cisco.com/performance/what-is-an-mpi-eager-limit. (cited on page 60)

Teranishi, K. and Heroux, M. A., 2014. Toward local failure local recovery resilience model using MPI-ULFM. In *Proc. 21st European MPI Users' Group Meeting*, 51. ACM. (cited on page 67)

The TLA Home Page. The TLA Home Page. http://lamport.azurewebsites.net/tla/tla.html. (cited on page 87)

Thoman, P.; Hasanov, K.; Dichev, K.; Iakymchuk, R.; Aguilar, X.; Gschwandtner, P.; Lemarinier, P.; Markidis, S.; Jordan, H.; Laure, E.; et al., 2017. A taxonomy of task-based technologies for high-performance computing. In *International Conference on Parallel Processing and Applied Mathematics*, 264–274. Springer. (cited on page 10)

Tipparaju, V.; Krishnan, M.; Palmer, B.; Petrini, F.; and Nieplocha, J., 2008. Towards fault resilient Global Arrays. *Parallel computing: architectures, algorithms, and applications*, (2008), 339–345. (cited on page 25)

Venkatesan, S., 1989. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38, 1 (1989), 103–110. (cited on pages 70 and 72)

Vinoski, S., 2007. Reliability with Erlang. *Internet Computing, IEEE*, 11, 6 (2007), 79–81. (cited on page 29)

Wang, C.; Mueller, F.; Engelmann, C.; and Scott, S. L., 2008. Proactive process-level live migration in HPC environments. In *Proc. ACM/IEEE conference on Supercomputing*, 1–12. IEEE Press. (cited on page 16)

X10 Applications. X10 applications. https://github.com/x10-lang/x10-applications. (cited on page 160)

X10 Benchmarks. X10 benchmarks. https://github.com/x10-lang/x10-benchmarks. (cited on page 148)

X10 Formal Specifications. TLA+ specification of the optimistic finish protocol and replication protocol. https://github.com/shamouda/x10-formal-spec. (cited on page 89)

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S.; and Stoica, I., 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX conference on Networked Systems Design and Implementation*, 2–2. USENIX Association. (cited on page 31)

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; and Stoica, I., 2010. Spark: Cluster computing with working sets. *HotCloud*, 10, 10-10 (2010), 95. (cited on page 17)

Zheng, G.; Shi, L.; and Kalé, L. V., 2004. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing, 2004 IEEE International Conference on*, 93–103. IEEE. (cited on page 28)

Zheng, Y.; Kamil, A.; Driscoll, M. B.; Shan, H.; and Yelick, K., 2014. UPC++: a PGAS extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 1105–1114. IEEE. (cited on page 24)