Resilient X10

Efficient failure-aware programming

David Cunningham[2] *, David Grove[1], Benjamin Herta[1], Arun Iyengar[1], Kiyokuni Kawachiya[3], Hiroki Murata[3], Vijay Saraswat[1], Mikio Takeuchi[3], Olivier Tardieu[1]

[1]IBM T. J. Watson Research Center
[2]Google Inc.
[3]IBM Research - Tokyo

dcunnin@google.com, {groved,bherta,aruni,vsaraswa,tardieu}@us.ibm.com, {kawatiya,mrthrk,mtake}@jp.ibm.com

Abstract

Scale-out programs run on multiple processes in a cluster. In scale-out systems, processes can fail. Computations using traditional libraries such as MPI fail when any component process fails. The advent of Map Reduce, Resilient Data Sets and MillWheel has shown dramatic improvements in productivity are possible when a high-level programming framework handles scale-out and resilience automatically.

We are concerned with the development of general-purpose languages that support resilient programming. In this paper we show how the X10 language and implementation can be extended to support resilience. In Resilient X10, places may fail asynchronously, causing loss of the data and tasks at the failed place. Failure is exposed through exceptions. We identify a *Happens Before Invariance Principle* and require the runtime to automatically repair the global control structure of the program to maintain this principle. We show this reduces much of the burden of resilient programming. The programmer is only responsible for continuing execution with fewer computational resources and the loss of part of the heap, and can do so while taking advantage of domain knowledge.

We build a complete implementation of the language, capable of executing benchmark applications on hundreds of nodes. We describe the algorithms required to make the language runtime resilient. We then give three applications, each with a different approach to fault tolerance (replay, decimation, and domain-level checkpointing). These can be executed at scale and survive node failure. We show that for these programs the overhead of resilience is a small fraction of overall runtime by comparing to equivalent non-resilient X10 programs. On one program we show end-to-end performance of Resilient X10 is $\sim$100x faster than Hadoop.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

Keywords X10, Distributed, Parallel, Resilience

1. Introduction

Scale out programs run on multiple processes in a cluster. In scale-out systems, processes can fail. They run out of memory, the node they are running on may lose power, overheat, or suffer a software failure.

Computations using traditional libraries such as MPI fail when any component process fails. If the mean time between failures (MTBF) of a single node is 6 months, a 24 hour job running on 1000 nodes has less than 1% chance of successful completion. Supercomputers are generally designed to be significantly more reliable than commodity clusters; nevertheless, resilience is a serious issue at extremely large scales.

Traditionally, long-lived multi-node applications have addressed node failure only via application-level checkpointing. This is *ad hoc*, error prone, problematic when the amount of state to be saved is large, and not applicable (or inefficient) in certain cases. For instance, for some machine learning applications a reasonable recovery strategy is to ignore the work assigned to the failed node and proceed to completion with the remaining nodes. We call this (checkpoint-free) strategy *decimation*. Depending on the algorithm, it may also be possible to proceed by recovering state from neighbors, from the original input files, through replay of work, or some combination of the above.

More recently, the advent of Map Reduce [12, 29], Resilient Data Sets [32], Pregel [17] and MillWheel [2] has shown the dramatic improvement in productivity possible

* Work done while employed at IBM T. J. Watson Research Center.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555248

when a high-level programming framework handles scale-out and resilience automatically. Nevertheless, the underlying programming model for which resilience is provided is very limited in these cases. Performance is a significant problem, and derivative frameworks such as M3R [25] improve performance but at the cost of loss of resilience.

We are concerned with the development of general-purpose, imperative languages that support resilient programming.[1] Over the last ten years, the X10 programming language [6, 22] has been developed as a simple, clean, but powerful and practical programming model for scale out computation. Its underlying programming model, the APGAS (Asynchronous Partitioned Global Address Space) programming model [23], is organized around the two notions of *places* and *asynchrony*. A place is an abstraction of shared, mutable data and worker threads operating on the data, typically realized as an operating system process. A single APGAS computation consists of hundreds, potentially tens of thousands of places. Asynchrony is provided through a single block-structured control construct, `async S`. If `S` is a statement, then `async S` is a statement that executes `S` in a separate thread of control (*activity* or *task*). Dually, `finish S` executes `S`, and waits for all tasks spawned (recursively) during the execution of `S` to terminate, before continuing. Memory locations in one place can contain references (*global refs*) to locations at other places. To use a global ref, the `at (p) S` statement must be used. It permits the current task to change its place of execution to `p`, execute `S` at `p` and return, leaving behind tasks that may have been spawned during the execution of `S`. The termination of these tasks is detected by the `finish` within which the `at` statement is executing. The values of variables used in `S` but defined outside `S` are serialized, transmitted to `p`, and de-serialized to reconstruct a binding environment in which `S` is executed. Constructs are provided for unconditional (`atomic S`) and conditional (`when (c) S`) atomic execution. Finally, Java-style non-resumptive exceptions (`throw`, `try/catch`) are supported. If an exception is not caught in an `async`, it is propagated to the enclosing `finish` statement. Since there may be many such exceptions, they appear wrapped in a `MultipleExceptions` exception.

The power of X10 lies in that these constructs can nest arbitrarily, almost without restriction[2]. A diverse variety of distributed control and communication idioms (e.g. recursive parallelism, active messaging, single process multiple data, accelerator off-load, remote memory transfers etc.) can be realized just through particular combinations of `async/at/finish/when`. Indeed, the usefulness of X10 for scale-out programming has been amply demonstrated. Large portions of the X10 runtime are written in X10. [28] presents benchmark results on over 50,000 cores, with performance comparable to MPI (in some cases significantly better). [25] develops a main memory implementation of Hadoop with significant performance improvements over Hadoop. [24] presents a novel, high performance algorithm for unbalanced tree search that leverages `finish`. [18] presents a large, sophisticated, computational chemistry code written in X10. [11] develops a graph library in X10. [10, 13, 14] present results on the productivity of programming in X10.

In this paper, we show how the X10 language and implementation can be modified to support resilience. Because of the orthogonality of the core X10 constructs, this is conceptually straightforward. A place `p` may fail at any time, with the loss of its heap and tasks. Any `at(p) S` executing at a place `q` will see a `DeadPlaceException` (`DPE`). Any attempt to launch an `at(p) S` at `p` will also throw a `DPE`. Global refs pointing to locations hosted at `p` now "dangle", but they can only be dereferenced via an `at (p) S`, which will throw a `DPE`.

However, X10 permits arbitrary nesting of `async/at/finish`. Hence when a place `p` fails, one or more tasks running at other places may be in the middle of executing an `at (p) S` and, conversely, code running at the dead place `p` may be in the middle of running `at (q) T` statements at other (non-failed) places `q`. What should be done about such cross-place dependencies?

Let u = `at (p) S` be a statement running at `r`, and `S` (running at `p`) contain a sub-statement v = `at (q) T`, for `p` $\neq$ `q`. We will call v a *non-local child task* of u. Note that v may have been launched within an arbitrarily nested `finish/async` sub-statements of u. What should be done with such "orphan" tasks when the spawning place (`p`) fails? We argue that it is unwise to track down and terminate such tasks. Asynchronous termination of tasks can leave the heap at different unfailed places in an inconsistent and unknown state. Tasks should either run to completion or not at all. However, we insist on a key design principle, the *Happens Before Invariance (HBI) Principle*:

> *Failure of a place should not alter the happens before relationship between statement instances at the remaining places.*

This guarantee permits the Resilient X10 programmer to write code secure in the knowledge that even if a place fails, changes to the heap at non-failed places will happen in the order specified by the original (unfailed) program. Failure of a place `p` will cause loss of data and control state at `p` but will not affect the concurrency structure of the remaining code.

As described below, this principle requires that the X10 runtime maintain some information about the control (`finish/async`) graph at every place in *resilient storage* (storage that outlasts the failure of a place). This information is used to repair the `finish` and `at` control dependency structure across places so as to correctly implement the HBI Principle.

[1] Erlang is an example of a general-purpose language supporting resilience that is not imperative.

[2] X10 restricts the `atomic S` and `when(c) S` constructs so that `S` may not dynamically execute `at (p) T`, `finish T` or `when (c) T` constructs.

Thus, Resilient X10 is obtained from X10 by permitting places to fail asynchronously, exposing failure through exceptions, and ensuring that the X10 runtime repairs the global control structure of the executing program to correctly implement the HBI Principle. In related work [8], we develop a formal semantics for Resilient X10 as a derivative of the formal semantics of X10. The semantics is presented as a transition system over configurations that are pairs $\langle s, g\rangle$ of a statement s (representing the tree of all tasks running in all places) and the global heap g. To model Resilient X10, this semantics need only be extended by a single rule that permits a place to fail asynchronously. g is changed to a g' that is the same as g except that all the state of p is lost. s is rewritten in a single step to a new statement s' that represents the loss of all tasks running at p and repairs the structure of the remaining tasks so that the HBI Principle (and an associated Exception Masking Principle) are preserved.

This paper presents the design and implementation of Resilient X10. Resilient X10 was implemented by modifying the X10 runtime and core libraries. Because the source language was unchanged, the X10 compiler is unchanged. The Resilient X10 programmer essentially writes X10 code, but with the awareness that DPEs may be thrown. User code may catch these exceptions and take appropriate recovery steps, based on domain specific information (e.g., discard lost state, reload from resilient storage, recompute, recover from neighbors, etc.). Note that a Resilient X10 program can run on an X10 implementation without change, and behaves just like an identical X10 program (such programs abort when a place fails). An X10 program can run on a Resilient X10 implementation without change, but may see DPEs on place failure. In summary, the contributions of this paper are:

1. The design and implementation of Resilient X10, which we believe to be the first implementation of a real, general-purpose, imperative programming language that allows applications to handle node failure without significant loss of scalability, performance, or productivity.
2. An evaluation of the overhead and scalability of our implementation for a variety of concurrency patterns.
3. A demonstration of the flexibility of Resilient X10 via benchmarks that handle failure in diverse ways. In particular, we identify decimation, an idiom that is applicable in many relaxed computing [5] settings, and which can be very simply and elegantly expressed in Resilient X10.
4. A comparison of sparse matrix dense vector multiply implementations showing that the end-to-end performance of Resilient X10 is $\sim$100x faster than Hadoop.

Section 2 continues with a more detailed comparison to existing work. Section 3 is a deeper discussion of the language design and programming idioms. Section 4 describes our implementation. Section 5 gives performance analysis and describes Resilient X10 applications, and Section 6 concludes.

2. Related Work

Hadoop [9, 29] handles failures invisibly, by writing *all* data out to a resilient disk store (HDFS) after each MapReduce job, and at key intermediate points. Hadoop, while relatively easy to program, is also restrictive in the types of algorithms it can support. Resilient X10 gives more flexibility to the programmer, both in how distributed programs are written, as well as giving the programmer control over how to cope with failures.

The Spark framework [32] uses Resilient Distributed Datasets, which provides a read-only collection of objects partitioned across multiple nodes. RDDs offer more flexibility and performance over Hadoop MapReduce, because all data does not need to be re-written to disk between each phase of the computation, and the programmer has more control over what data is persisted vs discarded. But Spark's programming model is much closer to MapReduce than a general programming language such as X10.

The Charm++ programming system [21] for distributed systems supports distributed termination detection as well as fault-tolerance via checkpoint/restart, but the user is responsible for combining the two safely. Typically checkpointing only happens at global synchronization points with no outstanding asynchronous processing. A similar approach, X10-FT, checkpoints the X10 state at the granularity of APGAS language constructs [31]. In contrast, Resilient X10 precisely defines the semantics of termination detection in the presence of failures. This makes it possible to continue executing in spite of failures without reverting to a checkpoint, while preserving the execution order of all surviving tasks.

Lifflander et al. [16] propose three fault-tolerant algorithms for detecting distributed termination for a core programming model and failure model similar to ours (asynchronous tasks, single root, fail-stop failures). They consider a single termination scope whereas we support multiple finish scopes (side-by-side or nested) thus providing more expressive power. In their algorithms, the shape of the spawn tree dictates the flow of control messages of the distributed termination detection as well as the layout of the redundant storage of the control state. A parent process is responsible for handling the failure of its children (possibly transitively), but the opposite is not possible. The memory overhead of fault tolerance is therefore not balanced across processes. In contrast, we decouple the resiliency implementation from the application structure. For instance, we provide an implementation where the task management state is centralized and another where it is distributed across all the nodes.

Containment domains [7] are a methodology whereby the programmer can provide fault tolerance in a modular fashion by choosing to either mask or report faults at the module boundary. In Resilient X10 it is easy for the programmer to follow this methodology. Masking errors means catching exceptions within a module and employing some internal re-

dundant application-level computation/storage to return the correct result despite the failure. If that is not possible, the error can be reported by either simply not catching the exception (which may expose details of the module's implementation) or preferably by catching it and rethrowing a more abstract version that better matches the module's API.

3. Language Design and Examples

A complete but simple Resilient X10 program, a Monte Carlo application that calculates π, is given in Figure 1. Execution begins at place zero. Because of this, place zero typically has a special role of communicating the result to the user, so we assume it can never fail (if it does fail, the whole execution is torn down)[3]. From there, the application is free to spawn tasks on other places. In this example, an asynchronous task is created at every place (Lines 9, 11), tries a number of random samples (Lines 14–18), and communicates its result back to place 0 using a global ref[4] (Lines 20–25). If a place dies during this work, its task throws a `DeadPlaceException` (`DPE`) which is silently caught (Line 26). The last few lines (Lines 28–30) will therefore execute when all the work is either completed or dead. If places die before they communicate back, then they contribute no result, and the accumulated number of tested samples is correspondingly lower. Thus losing a place will cause the computed value of π to be less accurate.

This simple example does not show-case the full expressive power of Resilient X10. In particular Resilient X10 is unique in allowing the arbitrary nesting of `finish` and `at`. This allows blocking for distributed termination within a task, which is not supported by Charm++ [21]. This allows two levels of concurrency for distributed multicore programming, and the writing of libraries that internally use `finish`. It also allows divide-and-conquer parallelism where each invocation of a recursive method waits for the termination of its recursive calls before returning control [19]. In the context of this rich concurrency, Resilient X10 attempts to make handling errors as easy as possible without compromising performance.

The Happens Before Invariance (HBI) Principle means programmers need not be concerned with subtle concurrency bugs due to failed synchronization. In the following three examples, place `p` fails during the execution of `S`.

In this example, Resilient X10 ensures that `R` is executed after `S`, as in regular X10:

```
try {
    at (p) { at (q) S; ... }
} catch (e:DPE) { } R
```

```
 1: import x10.util.Pair;
 2: import x10.util.Random;
 3: public class ResilientMontePi {
 4:   static val ITERS = 10000000 / Place.MAX_PLACES;
 5:   public static def main (args : Rail[String]) {
 6:     // (points_in_circle, samples)
 7:     val cell = new Cell(Pair[Long,Long](0, 0));
 8:     val cell_gr = GlobalRef(cell);
 9:     finish for (p in Place.places()) async {
10:       try {
11:         at (p) {
12:           val rand = new Random(System.nanoTime());
13:           var total : Long = 0;
14:           for (iter in 1..ITERS) {
15:             val x = rand.nextDouble(),
16:                 y = rand.nextDouble();
17:             if (x*x + y*y <= 1.0) total++;
18:           }
19:           val total_ = total;
20:           at (cell_gr) async atomic {
21:             // add our result to global total
22:             val the_cell = cell_gr();
23:             the_cell(Pair(the_cell().first+total_,
24:                           the_cell().second+ITERS));
25:           } }
26:       } catch (e:DeadPlaceException) { /* just ignore */ }
27:     } /* end of finish */
28:     val samples = cell().second;
29:     val pi = (4.0 * cell().first) / samples;
30:     Console.OUT.println("pi = "+pi+" (samples: "+samples+")");
31: } }
```

Figure 1. Computing π with the Monte Carlo method.

Adding `async` means `R` may execute in parallel with `S`. In Resilient X10, the termination of `S` is detected by the outer (implicit, not shown) finish as in regular X10:

```
try {
    at (p) { at (q) async S; ... }
} catch (e:DPE) { } R
```

Resilient X10 preserves the ordering guarantees from X10 by ensuring that `R` is executed *after* `S`, even though the `finish` governing `S` executes in a failed place:

```
try {
    at (p) { finish at (q) async S; ... }
} catch (e:DPE) { } R
```

In the examples above, what if `S` throws a user-defined exception? In regular X10 it would propagate to the closest `catch` block. However in the event of failure, the run-time representation of part of that control stack is now lost. We cannot tell whether the exception would have been caught in this lost part of the stack. Our options are: (1) Propagate the exception around the failure. (2) Drop the exception. (3) Make the control stack resilient. We decided that option 1 would be too surprising for programmers, since suppressed exceptions would never escape in regular X10. We chose option (2) to avoid the overhead and complexity of (3). The semantics of option (2) are actually quite reasonable. We give the programmer guarantees about synchronization, but we do not attempt to recover that task's exception output. A user-defined exception is never lost without there being at least one `DPE` appearing in its place, so the user-defined exception can be considered to be masked, rather than lost; we call this the *Exception Masking Principle*. Note that in the second example, the presence of the `async` means that any exception thrown by `S` will be routed directly to the outer finish, thus is not propagated through `p` and is not lost.

[3] This means the MTBF of the system is the same as the MTBF of a single node, which we consider to be acceptable given the prevalence of divide-and-conquer and the convenience of orchestrating and completing the execution at a single root node. We discuss this further in Section 6.

[4] In the `at` block, any outer scope variables may be *captured*. All captured variables are copied deeply to the target place. Global refs override copying to enable the creation of cross-place references.

In summary, loss of place p will cause loss of the heap at p, loss of (most of) the control state at p and also masks any exceptions that would have been propagated through p.

3.1 Resilient Storage

A resilient store is a data structure that internally stores data in a manner that survives node failure. This can involve replication in another node's memory or on disk. Storing in memory is faster, but has space overhead. However it is attractive because the application is self-contained. Either way has a performance overhead, so use of resilient stores should be limited to critical data that cannot be recovered by other means. Another useful application of resilient stores is infrequent checkpoints of application state.

Resilient in-memory stores can be easily implemented in Resilient X10. A class can be designed that provides a simple interface to read and write to the store. The implementation of that class will then update one or more backups behind the scenes. If a synchronous update is not required, the `async` construct can be used. This can provide better performance with fewer consistency guarantees. As in collection libraries, we expect that there will not be one resilient store for all potential uses. Instead there will be a few different stores abstracted into utility libraries, as well as applications occasionally implementing specialized versions with particular properties. Using the resilient storage, we have prototyped several resilient-aware libraries such as a resilient `DistArray`. The design, implementation, and usage of resilient data structures is an area of current research.

4. Implementation

Implementing Resilient X10 required significant work at all levels of the X10 runtime. Failure of remote nodes is discovered at the lowest level (the communication layer) and propagated up the stack. Special termination detection algorithms needed to be written in order to implement the HBI Principle.

4.1 X10RT

The X10 runtime has the ability to use one of several communications libraries, depending on network it is running on. A common API (X10RT) exists within the X10 runtime for abstracting the different libraries [30]. X10RT implementations exist that support running on top of TCP/IP sockets, shared memory, MPI, or PAMI [20].

Node failure is handled by current MPI and PAMI by ending the entire program. Therefore, it is not possible to constructively handle failure by building on top of MPI and PAMI. However, the sockets backend gave us the freedom to handle the failure of individual connections.[5] We therefore built Resilient X10 on top of sockets. We could support future implementations of MPI and PAMI if they provide appropriate failure notifications.

We extended the sockets implementation thus: Upon detection of a dead place (via a communication error or a configurable timeout), we clean up the link to that place, mark it as dead, and continue running. The message send API is asynchronous, which meant the failure to send a message could not be synchronously communicated up the stack. Instead, we added an API call to return the number of dead places (which never decreases) as well as calls to query the life/death status of arbitrary places. The typical use of this API by the X10 runtime involves checking whether the remote place has died while waiting for a message from that place. The runtime can thus tell if it is never going to receive the message and yield an error. The other X10RT backends (MPI, PAMI) continue to force an abort of the entire application upon the detection of a dead place, and so their implementations of the new methods never report a place death.

4.2 Finish

The `finish S` construct blocks until all tasks spawned by `S` have terminated. Since it is not statically known how many tasks there will be, where they will be running, and for how long, implementing finish requires a distributed termination detection algorithm. Regular X10 already has such an algorithm. Typically, an X10RT message from the spawning task to the *finish home* (the place on which the finish was created) records new tasks, and an X10RT message from the task's place to the home place is used to notify that a task has terminated. Often, global refs are used to reference the finish representation from a remote task.

A variety of finish implementations take advantage of particular patterns of concurrency. The default finish is optimized to locally cache updates and only update the home upon local quiescence. Other implementations are specialized for single remote tasks or SPMD-style workloads. However none of these implementations are resilient. If a place dies, these finish implementations forever wait for termination messages from the dead place.

We implemented 3 new finish implementations for Resilient X10. All are based on an abstract algorithm that maintains counters and can be described assuming its own state is resilient. The implementations differ in how the resilient storage is achieved.

Each implementation is realized with a *finish object* that is created at the opening brace of the finish. The ending brace of the finish is compiled to a `wait()` call on the object. Finish objects are used both for the *explicit* finish statements that occur in the code, and to implement the synchronous semantics of the `at` construct by spawning the remote task under an *implicit* finish. New tasks are governed by the closest enclosing explicit finish. When a finish object is copied across the network, the remote copy acts as a proxy that allows implicit communication but otherwise can be considered an alias of the original finish object.

[5] In sockets backend, 1-to-1 connections are established between each pair of communicating places.

```
public class FinishObject {
    public static def make(parent:FinishObject) : FinishObject;
    public def wait() : void; // may throw MultipleExceptions
    public def fork(src:Place, dst:Place) : void;
    public def begin(src:Place, dst:Place) : Boolean;
    public def join(dst:Place) : void;
    public def pushExc(e:Exception) : void;
}
```

Figure 2. Runtime API of a finish implementation.

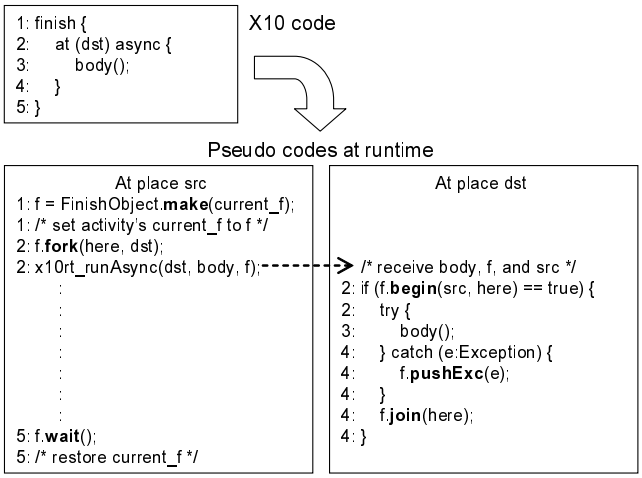

Figure 3. How the finish APIs are called.

The runtime maintains two stacks of finish states per task. The *synchronization* stack can be peeked to find the closest synchronization point. This is the closest implicit finish that is not outside of an `async`, otherwise it is the closest explicit finish. The *explicit* stack can be peeked to find the closest explicit finish, which governs new `asyncs`. In regular X10, only the explicit stack was needed. The synchronization stack is used for *adoption*, which will be explained shortly.

Any task can peek at either of these stacks to obtain the relevant finish object. As tasks are spawned and terminate, they call methods on the relevant finish object which internally communicates with the waiting task via X10RT messages and/or shared storage. The finish object API is shown in Figure 2.

Figure 3 shows how these APIs are used for executing an asynchronous task and waiting for its termination. When spawning a remote task[6], the finish object on the top of the explicit stack is used. The `fork()` method is called by the `src` place, and then an X10RT message is sent to create the remote task. Since the message is asynchronous, the `src` place then advances to the next program statement. When the X10RT message is received by the `dst` place, `begin()` is called, and if this returns `true` then the task is concurrently executed. If not, the message is silently discarded which happens when the source place died after transmission but before reception of the message. When the task terminates at `dst`, `join()` is called. A given finish implementation may, within these methods, communicate with the finish home place using more X10RT messages. If an exception from the task is uncaught, it is communicated just before the `join()` call, via the `pushExc()` call. If there are any such exceptions, they are combined into a `MultipleExceptions` object which is thrown from the `wait()` call after termination. In the case of an implicit finish, the `MultipleExceptions` can only contain a single exception, so this is unwrapped and thrown as normal, thus propagating the exception through the `at` construct.

Conceptually, the finish state objects encapsulate counters that record what tasks are running where. We call this the *live* counter set. It also stores the exceptions accumulated via `pushExc()`. The `wait()` call returns control only when the counters representing non-dead places are zero. The dead counters are used to generate `DeadPlaceException` (`DPE`) objects which are added to the user-generated exceptions. The space overhead of the live counter set is $O(n)$ in the number of places per each active finish object.

In addition to this, if `src` dies after calling `fork()` but before `dst` calls `begin()`, then it is possible the message was not completely transmitted and the task is lost even though `dst` is still alive. Thus, the finish implementation must record the messages in transit between each pair of places. Such messages can be lost if either `src` or `dst` dies. If `src` dies, the message is silently dropped, but if `dst` dies, a `DPE` is generated. This ensures we get one `DPE` per task that was running at the time of the place death. Conceptually this is a 2-dimensional matrix, but the representation need not be $O(n^2)$ in memory overhead since it is very sparse (the number of messages in transit is limited by the size of network buffers). We call these the *transit* counters.

If `src` dies after sending the message, but before `dst` calls `begin()`, it is possible the finish will terminate and the task could execute on `dst` after the termination of the finish, violating the happens-before relationship. This is why we have the `begin()` call return `false` in this case, to prevent the execution of a task when it has been assumed to be lost.

The above semantics are sufficient for properly handling and reporting place failure, so long as the finish home place does not die. In that event, we call the tasks under that finish (which may be at other places that are still live) *orphaned*. The *parent* finish, which is determined when each finish is created by peeking the synchronous stack, must not assume that all tasks under it have terminated just because its counters are zero, because these counters do not record orphaned tasks. To solve this problem, if a place dies and there were active finishes on that place, the closest parent finish that is still alive must *adopt* these orphaned tasks. Broadly, this means exceptions are discarded, and the counters (both live and transit) from the dead finish are merged into its own

[6] Local task creation is exactly the same, except `src` = `dst`

counters[7]. This work is done during the `wait()` call, before checking the counters for termination.

To handle the case where these adopted tasks die (due to further place failures), we must ensure the generated `DPE`s, like any other exceptions, should not make it to the adopting finish. This difference in behavior means we need two sets each of live and transit counters. One set of counters records the non-adopted tasks, and is used both for termination detection and to generate `DPE`s upon place death. The other set of counters records adopted tasks and is only used for termination detection.

The orphaned tasks themselves must send subsequent finish updates to the adopting finish rather than to the dead finish, and these operations will modify the adopted counters. This can be achieved by setting a flag on the adopted finish during the adoption process, and leaving a forwarding reference. Any updates that must occur after the finish has died but before it has been adopted by the parent finish can simply update the dead finish state as usual. Both adopted and non-adopted counters from the dead finish are merged into the adopted counters of the adopting finish. To minimize the overhead, there is no built-in support for a user-level task to know whether it is adopted or not. However, if such knowledge is needed in specific cases, it could be implemented at the application (or framework) level on top of the basic primitives provided by Resilient X10.

In summary, a resilient finish implementation must record the tasks running directly under the finish, as well as adopting tasks from dead child finishes. It must record user-generated exceptions and generated `DPE`s for its immediate child tasks, while dropping them from adopted child tasks. The space overhead for a finish implementation is $O(n)$ assuming the transit matrix remains sparse. Having described how tasks are to be managed in the context of failure, we now describe 3 ways of using resilient storage to maintain the finish states.

4.2.1 Place-Zero-Based Finish

The simplest way to ensure that the finish state survives place failure is to store everything at place 0, which is assumed to never fail. This means every finish operation initiated by a place other than 0 involves a synchronous communication to 0. This is a bottleneck and requires more communication. However it is simple and performs reasonably well up to hundreds of nodes in common cases.

Place 0 contains a database of every finish state in the execution. Each finish state has a field indicating the home place and a field pointing to the parent finish, as well as all the counters. When place 0 discovers some other place has died, it scans the whole list, looking for finishes on the dead place. Each that it finds is adopted up to the nearest non-dead finish, by chasing the `parent` field.

[7] Note that the states of finish objects are stored in a resilient storage and are accessible even after the finish home place is dead.

We experimented with a simple 2-D array for the transit matrix, and a sparse representation based on the standard X10 standard library hash map. At the scales measured, the space overhead was negligible in both cases. Finish states are removed from the database (allowing garbage collection) when they terminate.

4.2.2 ZooKeeper-Based Finish

To provide a more scalable finish implementation, we tried to leverage ZooKeeper [15] to implement the required resilient storage. This seemed reasonable since ZooKeeper is touted for reliable task management in a cluster environment. Like a file system, ZooKeeper exposes a hierarchical database where each *znode* holds arbitrary data. In the ZooKeeper-based finish, all finish states are stored within ZooKeeper so can survive the death of places without depending on place 0 and should be more scalable.

X10 is implemented via translation to either Java (*Managed X10*) or C++ (*Native X10*). Currently, communication with the ZooKeeper server is implemented using Managed X10's Java interoperability framework [26, 27]. Therefore, the ZooKeeper-based finish is available only for Managed X10. The Native X10 implementation is left as future work.

Naive znode mapping: We considered two ways of mapping a finish state to znodes. The first method represents each counter of a finish state as a znode which holds the integer value, as shown in Figure 4(a). For each FinishState, a znode `FinishState-`*ID* is created, which holds various information for the finish state such as home ID, parent finish state ID and adopted flag. Under this FinishState znode, znodes `active` and `transit` are prepared, each of which contains znodes which hold corresponding counter values.

For example, a znode "`transit/0to1`" contains the number of tasks being created by place 0 at place 1 and a znode "`live/1`" contains the number of tasks being executed at place 1 under this finish. To reduce initialization overhead, these counter znodes are created dynamically when they are first used. Since the counter value may be modified in parallel from multiple places, a special znode "`lock`" is prepared for mutual exclusion. We use a lock mechanism provided in [3] with some modifications. Another znode "`excs`" is used to hold exception information.

The counter znodes are incremented and decremented by contacting the ZooKeeper server in each of the finish API methods. The `wait()` call blocks until all the counter znodes become zero. This logic is implemented by utilizing the *Watcher callback* mechanism of ZooKeeper. We evaluated the naive znode mapping and discovered it to be 50 times slower than the place zero implementation when creating an task.

Optimized znode mapping: In the naive znode mapping, the processing of `fork()`, `begin()`, and `join()` needed *multiple* ZooKeeper operations, each of which needs synchronous communication with the ZooKeeper server(s). To

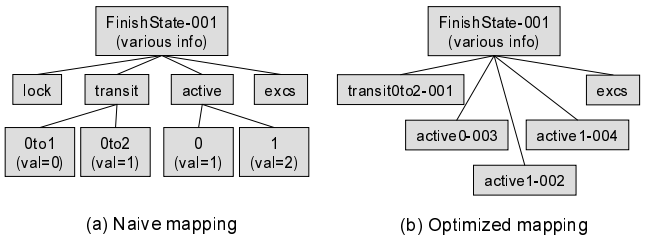

Figure 4. Mappings of a finish state to znodes.

ZooKeeper operation	Time
`create`	0.56 msec
`create` (with SeqNo)	0.58 msec
`delete`	0.33 msec
`exists`	0.34 msec
`getData`	0.49 msec
`setData`	0.57 msec
`lock`	1.43 msec
`unlock`	0.34 msec

Table 1. Cost of ZooKeeper operations.

minimize the overhead in ZooKeeper-based finish, we implemented another mapping of finish state optimized for ZooKeeper. This variation utilizes a ZooKeeper function that creates a znode with unique sequence number, enabled by specifying "`CreateMode.PERSISTENT_SEQUENTIAL`" for `ZooKeeper.create`. As shown in Figure 4(b), counter values are represented as the *number* of znodes rather than held in the znodes.

In `fork(S,D)` processing, a znode "`transitStoD-`*SeqNo*" is created, where *SeqNo* is assigned by ZooKeeper server. The sequence number is passed to the destination place D along with the task being created. In `begin(S,D)` processing, a znode "`liveD-`*SeqNo*" is created, then the passed znode "`transitStoD-`*SeqNo*" is deleted. In `join(D)`, the znode "`liveD-`*SeqNo*" is deleted. Since unique sequence numbers are assigned by ZooKeeper server, lock operations are not necessary for these processings. The `wait()` call waits until all of the `transit` and `live` znodes under the finish state are deleted. This blocking operation is also implemented by watching for znode deletion through the Watcher mechanism. Through this new mapping, the number of ZooKeeper operations necessary to spawn a task was reduced to 4 (`create` for `fork()`, `create` and `delete` for `begin()`, and `delete` for `join()`).

When we benchmarked this implementation, we discovered it was considerably faster than the naive mapping. However, it was still 13 times slower than the place-zero-based finish. To identify the reason of this overhead, we implemented a ZooKeeper-specific microbenchmark to measure the cost of ZooKeeper operations in our testing environment. The numbers were measured on the system described in Section 5 using IBM J9 Java VM [4] for Linux x86_64, version 7.0 SR6. We used ZooKeeper server version 3.4.5. To obtain a bound on the best possible performance we ran ZooKeeper on the same machine as the X10 program and used a ram disk for the snapshot storage of ZooKeeper. Table 1 summarizes our results.

Our experiments showed that each ZooKeeper operation takes 0.3 to 1.4 msec. The result is consistent with the numbers shown in a ZooKeeper paper [15], which states that the latency of a synchronous `create` operation is 1.2–1.4 msec. In our most efficient mapping to znodes, 4 ZooKeeper operations (2 `create` (with SeqNo) and 2 `delete`) are necessary for spawning one task. The total cost of these operations are 1.82 msec, which indicates that the slowness of ZooKeeper-based finish is a directed result of the cost of ZooKeeper operations.

For sufficiently large tasks, these overheads do not matter. However many X10 programs use fine-grained concurrency and small messages. We believe that, as a general-purpose resilient disk-backed database, ZooKeeper is too heavyweight to be the basis of termination detection in X10. After this experience, we decided to implement our own resilient store with the performance characteristics we needed.

4.2.3 Distributed Resilient Finish

The final finish implementation is an attempt to improve on the place-zero bottleneck by using an X10-level resilient storage for the finish state instead of relying on the implicit resilience of place 0. While this requires storing all the state in more than one place, this extra cost should be a constant factor overhead, and it can be done in a distributed manner that scales well. Moreover, implementing the resiliency within X10 avoids any out-of-process overheads and allows us to influence performance by controlling the manner in which the data is stored. However creating a resilient store within a finish implementation is harder than described in Section 3.1, because it must be built on top of X10RT messaging primitives instead of the high level APGAS primitives.

The basic idea is to store the finish at its home place, with a backup at some other place (more backups could be used for more resiliency but also more cost). If the home place happens to be place 0, then there is no need for a backup, and the implementation behaves like the place-zero-based finish. Otherwise, another place is chosen to contain a synchronized replica of the finish state. If both the master and the backup die, there is a fatal system-wide error. Otherwise, the finish implementation will transparently handle the fault. Currently, the next place is chosen for the backup, but more sophisticated algorithms could be used based on some model of failure correlation. The backup is accessed only to (1) receive updates from the master to keep it in sync, (2) receive updates from child tasks after the master had died but before adoption, (3) facilitate adoption. Every operation on the master internally performs a synchronous communication with the backup. This adds extra latency to the operation but en-

sures that the backup contains the most up-to-date information.

Because the backup is only used for termination detection if the finish home place dies, the backup need only store the data required to allow adoption by another finish. The backup therefore need not distinguish between regular and adopted counters, so to save memory it stores the pointwise sum of the master's two counter sets. After adoption, the backup is tagged with a forwarding pointer and is essentially inert. Another optimization is local tasks need not be backed up. This is because they die with the finish, so they never need to be adopted.

Each place has a *backup* table that can be used to find the backup of a given master if that master is dead. The table maps the master identity (the global ref to the master) to the backup object. Thus, if some task fails to communicate with the master because that place is dead, it can instead find the backup by searching other places' backup tables. The search can be accelerated if the backup place is chosen using a deterministic algorithm. This is an example of a global ref to a dead place being useful even though it cannot be dereferenced. The global ref still has a hash code and can be tested for equality with other global refs; thus it can be used to index a hash map. The fatal error for loss of master and backup arises when an exhaustive search (every place) does not find the backup.

With the distributed implementation, care must be taken when the master and backup are lost and a fatal error needs to be raised. Storing a list of child finishes at each finish (instead of a single parent pointer) means that even if master and backup disappear, the dangling pointer from the parent will indicate something has gone wrong. If a place dies, each finish checks if any of its child finishes were on that place. If so, the child pointer is used by the parent to find the backup and adopt the tasks. If a backup cannot be found, the fatal error is raised.

The finishes in the system thus form a distributed tree of masters and backups, with tasks referencing the masters, as shown in Figure 5. If a master is lost, one can still use the master reference to access the backup through the backup table. Thus the tree remains implicitly connected despite the loss of either M_n or B_n (but not both) for all n. When a parent finish adopts a child finish, it modifies its own list of child finishes by replacing the dead finish with the dead finish's child finishes. This process is repeated until all child finishes are alive, thus eliminating the dead finishes from the tree and ensuring there is no single point of failure. For example, if M_2 were to die, the tree would be modified such that M_2's tasks were governed by M_1 and M_1 would have M_3 and M_4 as child finishes.

5. Evaluation

We evaluate the practicality of the language design by writing one microbenchmark and three benchmark applications,

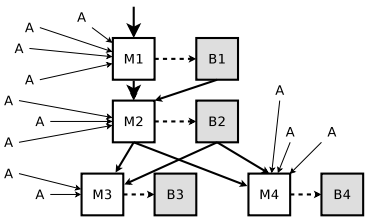

Figure 5. Distributed resilient finish tree.

each with a different approach to achieving resiliency. The microbenchmark tests the overhead of our X10-based finish implementations (compared to the regular algorithm). The applications show end-to-end running time and scalability, with and without resiliency.

Our benchmarking system is a 23 node AMD64 Linux cluster, each node having 16G RAM and 2 quad core AMD Opteron 2356 processors. The nodes are linked by gigabit ethernet, and are divided between two bladecenters. In this section we use Native X10 (the X10 C++ backend).

5.1 Microbenchmark

Just like X10, Resilient X10 supports arbitrary nesting of APGAS constructs. This makes the implementation challenging and can require significant runtime bookkeeping to orchestrate the running tasks and preserve the HBI Principle during failures. Therefore it is necessary to investigate both the constant overheads of this approach and identify potential scalability problems. To measure the overhead we wrote a number of microbenchmarks, each of which stresses the finish implementation in a different way. The microbenchmarks contain empty tasks, so only the termination detection overhead is measured. We tested many patterns but the differences we found can be illustrated with 3 of these in particular, shown in Figure 6.

The left benchmark spawns a single task at each place under a single finish (fan-out), and each of these tasks sends a single message back to the finish home place. This is of interest because it is a common pattern in our applications, e.g., Figure 1. The middle benchmark is a fan-out where each task creates its own local finish that governs 100 local tasks. This pattern occurs when using two levels of concurrency to both scale out and make use of multicore architectures. The right benchmark is a fan-out with each task doing another fan-out within a nested finish. This last case involves n^2 tasks and shows the non-scalability of the place-zero-based finish.

To demonstrate scalability on a machine that is closer to a typical supercomputer, we also ran these benchmarks on 13 node, 32 core per node *Power775* cluster, enabling a max of 416 places. These results are shown on the bottom row of graphs in Figure 6. We used the MPI backend of X10RT to take advantage of the cluster's very fast interconnect. This was possible because we were not testing failure, only overhead (the underlying HPC network library does not support

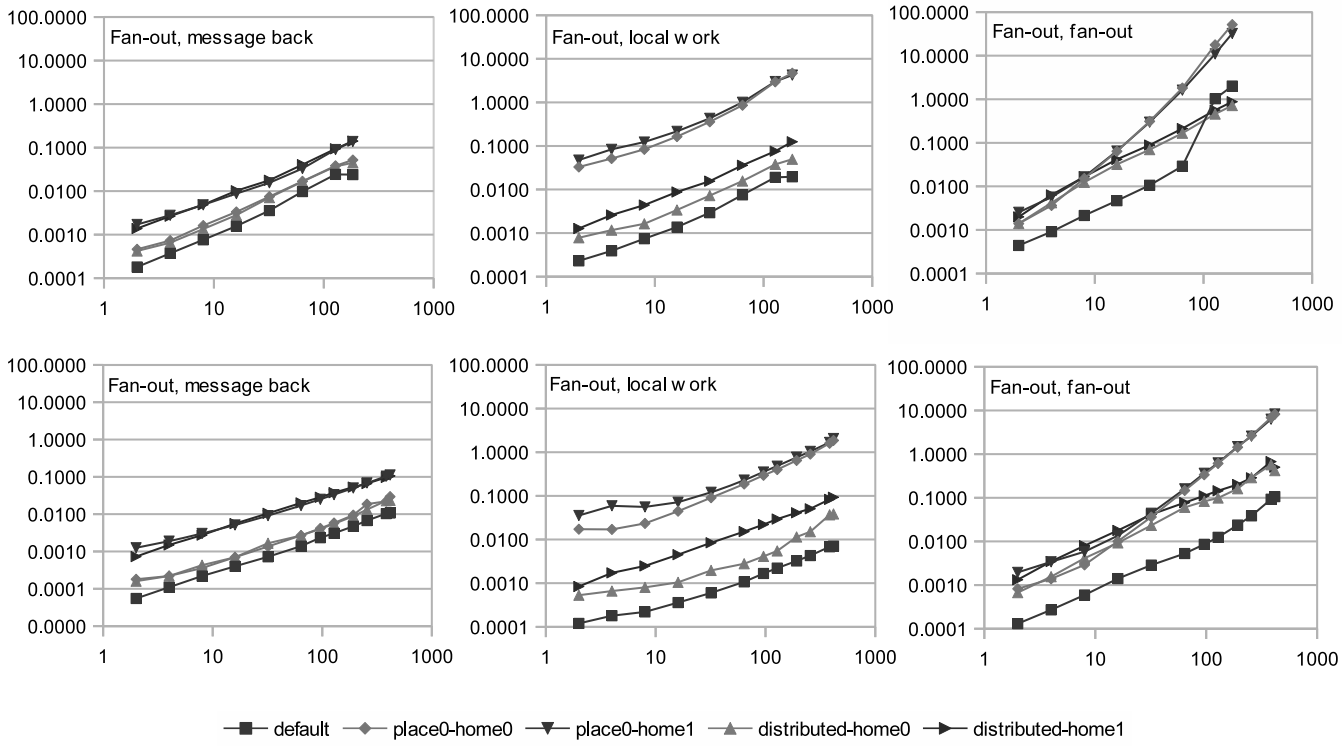

Figure 6. Scalability of the finish implementations, for different concurrency patterns. The top row of graphs are results from our 23 node AMD64 Linux cluster (max 184 places); the bottom row of graphs are results from a 13 node IBM Power775 cluster (max 416 places). The X axis is the number of places, the Y axis is the execution time in seconds.

failure). We hope that our work on Resilient X10 will motivate the development of HPC network libraries that handle failures so we can use them in the future.

We ran 5 experiments per benchmark. We compared the non-resilient (default), the place-zero-based and the distributed resilient finishes. Since the resilient implementations have different behaviors depending on whether the home is 0 or some other place, we ran the tests based at both place 0 and place 1. The results expose some qualitative differences between the implementations.

Firstly, there is a cost to resiliency. This is because resilient finish implementations require more synchronous communication because failure can preempt the execution at any time. In the left hand case, the two resilient implementations are similar in performance, but the need to store state in a resilient store (when finish is based outside of place 0) costs an order of magnitude in performance. In the middle case, the weakness of the place-zero-based implementation becomes clear: The distributed implementation is optimized to not store local tasks in the backup. However, the place-zero-based implementation must always communicate with place 0 even when managing local tasks. The final benchmark shows the place-zero-based implementation scaling very poorly because n^2 messages contend at place 0.

Exploiting our base assumption that place 0 will not fail, some communication is avoided for finishes based at place 0. This is why "home0" configurations showed slightly better performance than "home1" (i.e., non place 0 home). However, putting more work in place 0 can easily become a scalability bottleneck. It also depends on the application whether the `finish`es can be scattered among places.

In summary, there is measurable but manageable cost to resiliency. The place-zero-based implementation performs well enough in some cases but in general the distributed implementation is faster and more scalable.

5.2 Iterative sparse matrix dense vector multiply

This application represents a kernel found in a diverse range of analytics applications, including GNMF (Gaussian non-negative matrix factorization) and page rank. It was particularly interesting to us, because we had access to a Hadoop implementation of this algorithm that performed well within the Hadoop framework. This allowed direct comparison of Resilient X10 performance with Hadoop.

The input is an $N \times N$ sparse matrix (0.1% of elements are non-zero) of doubles, G, and a dense vector of N Doubles, V. The algorithm computes $U = G \times V$ and then U is used in place of V in the next iteration. Thus, the matrix remains constant for the whole execution but the vector con-

verges to the end result. For benchmarking purposes we run for a fixed number (30) of iterations instead of testing convergence. Our input is randomly generated between 0 and 1. Both matrix and vector are blocked with a factor of 1000.

Since Hadoop only offers end-to-end (job) timings and does not separate disk I/O from other overheads, we implemented a full application capable of reading the input matrices and writing the resulting vector to disk. We used GPFS instead of HDFS, since GPFS manifests as an OS-level filesystem and thus the data was accessible using X10's standard I/O library. Rather than parse the file metadata from the Hadoop SequenceFileFormat, we wrote a Hadoop program to write out the data to a simpler format that we then read in the X10 program. The metadata is a negligible proportion of the file size so this does not affect our results.

This is not a computationally dense benchmark. During the matrix multiply, each matrix element is read from memory and has only one multiply-add instruction performed on it. Yet it is desirable to use more than one node because G will not fit in the memory of a single node. Therefore G is partitioned into $N/1000$ row blocks, and the multiplication work associated with these row blocks is divided across the available nodes as evenly as possible. The initial V is loaded from disk by place zero at initialization time. At each iteration, the places load any fragment of G they need but have not already loaded. Ignoring failures, this means G is loaded only once at the first iteration. When failures occur, the work assigned to dead places is reassigned, so places will load these new parts of G when they are first needed. When G is known to be ready, the input V is broadcast to all nodes. Each place uses all of V and its fragments of G to compute corresponding fragments of U. Finally the fragments of U are then concatenated at place zero to become the next iteration's V.

When distributed in the above manner on our cluster, the algorithm is network-bound as each place receives a full copy of V each iteration. For this reason we chose to use one worker thread and one place per node. There is not enough computational work to justify using more cores, and creating more places per node would just mean sharing RAM and network bandwidth. For the smallest data sizes, it may be possible to do the whole computation on one node to get better absolute performance. However we are primarily interested in investigating the scalability of the computation when the matrix (G) cannot fit in one node. Therefore all measurements are performed with the data distributed.

When a place fails, we lose its portion of G, the input V and its partially computed U. The loss of V is immaterial since that is duplicated on every other place. The partial loss of G is not a problem since it is never modified, and the lost fragments still exist on disk. The only real loss is the partial U which cannot be recovered without replaying that work.

Our implementation responds to place failure as follows: Any/all `DPE`s are caught together at the finish block for that iteration. If there are any failures, the work assigned to the dead places is reassigned, all U are discarded and the last iteration is replayed with the new work assignment (missing out the dead nodes).[8] The execution then continues for the remaining iterations. Since G is now spread across fewer nodes, each place must have spare RAM to accommodate place failures. The cost of a failed place can be broken down: The previous iteration is lost, the remaining places must load their newly assigned parts of G from disk. Finally, each future iteration is slower due to each place having more work to do. Thus, the cost of place failure depends on the number of failures and when they occur.

Size	Hadoop	X10	Res. X10	1 Dead Place
100K	3301	12	12	14
200K	3390	20	19	20
400K	3563	32	30	32
800K	5392	73	70	76
1M	6820	96	96	105
1.2M	8737	128	129	142
1.5M	12559	180	182	199
2M	21773	293	290	317
2.5M	33664	434	438	480
3M	52075	596	595	656

Table 2. End-to-end duration (seconds) for 30 iterations of sparse matrix dense vector multiply over the 23 nodes (23 places). In the final column, a place was killed at iteration 15; the longer execution time reflects the recovery and completion of the program with fewer nodes.

In order to further improve the performance of the Resilient X10 version, we implemented the broadcast of V each iteration using a broadcast tree instead of naive iteration to each place. One must avoid dead places, so we modify the tree to avoid routing through places that have no work assigned. Our solution was simply to delete nodes of the tree and merge their children into the parent. A method that preserves the depth of the tree is future work.

In Table 2, we give the end-to-end times for various matrix sizes, running on our full cluster (23 nodes, 23 places). Hadoop is on average 100x slower than X10, due to excessive disk I/O. The cost of Resilient X10 as opposed to regular X10 is lost in the noise, and the cost of recovering from the a single node failure half way through execution is 10%.

In the case of size 3M, the Resilient X10 end-to-end time (596 seconds) breaks down into 273 seconds reading the matrix form disk, and 11 seconds per iteration. Clearly if the number of iterations were increased, the performance gap between X10 and Hadoop would increase. The 536 lines of code in this benchmark break down as follows: I/O: 33%, matrix/vector block data structures and multiply:

[8] Partial replay for faster recovery can be implemented, but we focused here on the non-failure overheads. Our long term vision is that such details would be handled by various frameworks depending on the resiliency model.

29%, distribution/concurrency control: 21%, command line options: 11%, failure handling: 6%.

We hope these results show that while Hadoop can be a good choice for certain large data traversals, it is not suitable for general-purpose cluster programming.

5.3 K-Means

The K-Means benchmark is a distributed implementation of Lloyd's algorithm. The problem is to find the k centroids that approximate the distribution of n points, where n is much larger than k. Arbitrarily, we chose our points to be $4D$ and represented with Floats.

The algorithm starts with guessed positions for the k centroids, and iteratively refines them. Each iteration, each centroid's new position is calculated by averaging the points for whom that centroid was closest. The algorithm is distributed by splitting the points evenly across places, and replicating the cluster positions across every place. Each iteration, each place uses its local points to calculate a partial average for the new cluster locations, and these are then aggregated to form the new clusters for the next iteration.

We chose this problem because it allows an approach to resilient programming that we call *decimation*. If a place dies, we simply use the remaining state to continue executing. This makes sense because in analytics, the input points are not precise, and likely to already be a sampling of a real phenomenon. The loss of an arbitrary but small percentage of that input should therefore yield an equivalent result. Other algorithms that operate on a sampled input dataset would also apply, such as any Monte Carlo problem.

The benefit of decimation is that error recovery is essentially instantaneous and the running time is unaffected by failures. If the input data is ordered in any meaningful way then loss of a contiguous portion of it could yield a substantially different result, but this can be fixed by storing the data in a random order. If failures are anticipated then the amount of input data can be over-provisioned to control the error bounds.

Implementing the decimation technique in Resilient X10 means catching and ignoring any `DPE`s that arise, and modifying the output to inform the user that the result is approximate due to place death. This requires only a few lines of code to be added to a regular implementation of K-Means in X10. We show the performance of the implementation in Figure 7. We execute the implementation on Resilient X10 (using the place 0 implementation of finish) and on regular X10. The measurements were done for 1 to 184 places allocated on the 23 nodes in a round-robin manner. The scaling is close to linear for the non-resilient case, and about 10% slower with the Resilient X10 runtime. The performance is not affected by failures.

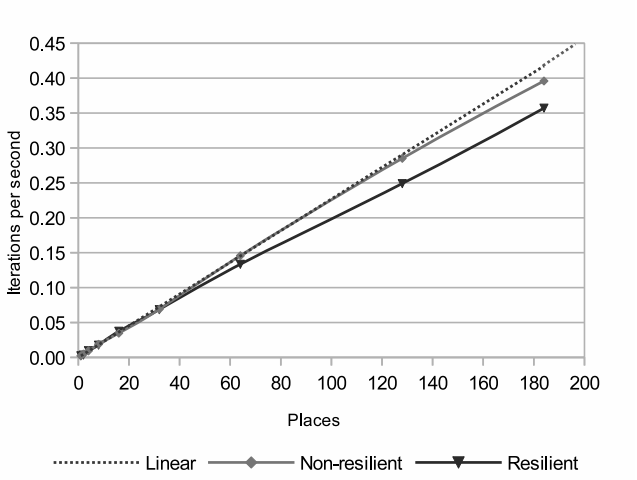

Figure 7. Strong scaling of K-Means ($n = 184000000$, $k = 100$), for 1 to 184 places over the 23 nodes.

5.4 Resilient Heat Transfer

The heat transfer application computes the diffusion of heat through a two-dimensional grid (represented with an array of Double). The algorithm iteratively updates heat values by averaging the values from neighboring points (i.e. by performing a stencil computation – a common HPC pattern). Usually the computation is run until convergence but for benchmarking purposes we ran a fixed number of iterations.

The array of heat values is distributed across multiple places. Each place contains a *front* and *back* array. Each iteration, one of them is assigned to, while the other stores the last iteration's result. These arrays are 2 elements bigger in each dimension, so that they can store the *skirt*, i.e. the values they need from their neighbors. The skirt is updated via communication between places, each iteration.

While the matrix multiply benchmark had a large amount of state, most of it (G) was not updated during execution. This allowed recovery to proceed by reloading it from the original input files. Heat transfer also has a large amount of state, but this state is frequently updated during execution. A different approach to resiliency is thus required.

For resilience, the heat values array is periodically checkpointed into a resilient store. We chose a single backup in a neighboring place's RAM. This is easier and faster than disk I/O and means the application is self-contained, but has more memory overhead. Other variations are possible.

Upon failure, all state outside the resilient store is discarded. The data is recovered from the resilient store and redistributed according to a new distribution over the remaining nodes. Execution then continues from the last checkpoint iteration. The frequency of checkpointing can be customized. Depending on the speed of the network, the problem size, and the expected frequency of failures, it is possible to tune the checkpoint frequency to give the best end-to-end time. Infrequent checkpoints reduce checkpointing overhead, but require more CPU time to recover from failure.

In order to survive failure during the checkpointing operation, it is necessary to keep both old and new checkpoints

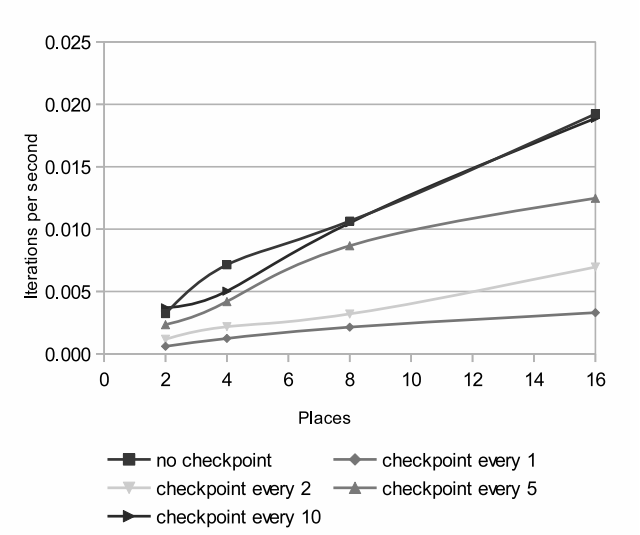

Figure 8. Strong scaling of heat transfer (16384×16384 grid) from 2 to 16 places with a variety of checkpointing frequencies.

in RAM until every place has finished checkpointing. This avoids the case where a place dies leaving its checkpoint not updated, but other places have completed their checkpoint. Recovering from that situation is impossible since there is no complete set of data for any given time point. This situation is analogous to using a disk storage, where the last checkpoint is not deleted until the new checkpoint is completely written. With our implementation, the space required is 3x that of the non-resilient version.

The stencil kernel is computationally non-intensive, so to avoid sharing the network interface we use 1 place per node. In Figure 8 we show the benchmark scaling to 16 nodes (16 places)[9] with a variety of checkpointing frequencies.

6. Conclusions and Future Work

We have designed and implemented an extension to X10 that allows general-purpose programming in an environment where node failure is common without compromising performance, scalability or productivity. We have made our implementation publicly available as part of the X10 2.4.1 open source release [1]. Resilient X10 preserves the happens before order of the original program, greatly simplifying the burden of handling node failures, while retaining performance transparency and allowing the programmer to use domain-specific knowledge to efficiently code resilient applications. We have measured the additional overhead of Resilient X10 in isolation, and found it to be modest, as well as scalable up to 416 places.

[9] Partitioning of the grid into equal squares is much harder for 23 nodes. To avoid additional communication overhead in the stencil algorithm by worse partitioning, we measured only 2, 4, 8 and 16 places.

We have described three ways of writing resilient applications in X10. Replaying from disk is appropriate when there is a large amount of immutable state that can be recovered from the original input file. Decimation is appropriate for applications where an approximate result is acceptable; it is the simplest to achieve and also the fastest. Finally, if there is a large amount of mutable state then a resilient store can be implemented in Resilient X10 to allow state to be checkpointed in memory at neighboring nodes. Resilient X10 makes implementing such stores much easier than other languages, and they can also be provided as a utility library.

In the future, we would like to construct frameworks on top of Resilient X10 for transparent resilient programming in specialized programming models. A wide range of frameworks are possible: MapReduce, bulk synchronous parallelism, matrix processing, etc. Each should be easy to write on top of Resilient X10 since it provides a lot of the basic functionality required. Research into brand new paradigms for transparent resilient programming can also be conducted on top of Resilient X10.

We would also like to revisit our immortal place 0 design. One option is to decentralize the execution, allowing the code to start at every place and co-operate in a peer-to-peer fashion. Another option is to make place 0 transparently resilient (by synchronously checkpointing all the program state). This would allow relocating it upon failure, but would also cripple its performance. However that would not be a problem if the only work at place 0 is initialization, coordination and communication with the user. Extending basic distributed classes such as `DistArray` to make them resilient is also underway.

Finally, we would like to support adding new places during the execution as well as detecting failed places. This would allow nodes to be replaced, e.g. for online maintenance of highly available distributed applications such as web servers and databases.

Acknowledgments

We would like to thank Avraham Shinnar for his advice regarding the Hadoop comparison, Josh Milthorpe for helpful discussions about the fast multipole method and Silvia Crafa for discussions about semantics.

This work was funded in part by the Air Force Office of Scientific Research under Contract No. FA8750-13-C-0052.

References

[1] X10 web site, 2013. URL `http://x10-lang.org/`.

[2] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*, pages 734–746, 2013.

[3] Apache Software Foundation. ZooKeeper Recipes and Solutions, 2012. URL `http://zookeeper.apache.org/doc/-current/recipes.html`.

[4] C. Bailey. Java Technology, IBM Style: Introduction to the IBM Developer Kit: An overview of the new functions and features in the IBM implementation of Java 5.0, 2006. URL `http://www.ibm.com/developerworks/java/library/-j-ibmjava1.html`.

[5] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 169–180, 2012.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[7] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *the Proceedings of SC'12*, November 2012.

[8] S. Crafa, D. Cunningham, V. Saraswat, A. Shinnar, and O. Tardieu. Semantics of (Resilient) X10. Dec. 2013. URL `http://arxiv.org/abs/1312.3739`.

[9] D. Cutting and E. Baldeschwieler. Meet Hadoop. In *O'Reilly Open Software Convention*, Portland, OR, 2007.

[10] C. Danis and C. Halverson. The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. *Third Workshop on Productivity and Performance in High-End Computing*, page 11, 2006.

[11] M. Dayarathna, C. Houngkaew, H. Ogata, and T. Suzumura. Scalable performance of ScaleGraph for large scale graph analysis. In *HiPC*, pages 1–9. IEEE, 2012.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic. An Experiment in Measuring the Productivity of Three Parallel Programming Languages. In *P-PHEC workshop, held in conjunction with HPCA*, February 2006.

[14] C. Halverson, C. B. Swart, J. P. Brezin, J. T. Richards, and C. M. Danis. The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. *1st International Workshop on Software Engineering for Computational Science and Engineering*, 2008.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11, 2010.

[16] J. Lifflander, P. Miller, and L. Kale. Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, February 2013.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. . URL `http://doi.acm.org/10.1145/1807167.1807184`.

[18] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove. X10 as a Parallel Language for Scientific Computation: Practice and Experience. In *IPDPS*, pages 1080–1088. IEEE, 2011.

[19] J. Milthorpe, A. P. Rendell, and T. Huber. PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[20] PAMI Guide: `http://tinyurl.com/pamiguide`.

[21] Parallel Programming Laboratory. The Charm++ Parallel Programming System Manual. Technical Report version 6.4, Department of Computer Science , University of Illinois, Urbana-Champaign, 2013.

[22] V. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367. Springer-Verlag, 2005.

[23] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing (colocated with PLDI 2010)*, Toronto, Canada, June 2010.

[24] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, 2011.

[25] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, Aug. 2012.

[26] M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, and T. Onodera. Compiling X10 to Java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, pages 3:1–3:10, 2011.

[27] M. Takeuchi, D. Cunningham, D. Grove, and V. Saraswat. Java interoperability in Managed X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, pages 39–46, 2013.

[28] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at Petascale. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, February 2014.

[29] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

[30] X10RT API: `http://x10.sourceforge.net/x10rt/`.

[31] C. Xie, Z. Hao, and H. Chen. X10-FT: transparent fault tolerance for APGAS language and runtime. In P. Balaji, M. Guo, and Z. H. 0001, editors, *PMAM*, pages 11–20. ACM, 2013.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, 2010.