Writing Fault-Tolerant Applications Using Resilient X10

Kiyokuni Kawachiya

IBM Research - Tokyo
5-6-52, Toyosu, Koto-ku, Tokyo 135-8511, Japan
<kawatiya@jp.ibm.com>

Abstract

X10 is a programming language that internally supports distributed computing. X10 applications can run over multiple “places” (computing nodes), and perform distributed computing by changing the execution place using “`at`” statements. However, in a conventional X10 environment, when a node that handles a place fails, the entire processing of that X10 application is aborted. To address this problem, we have been extending X10 to “Resilient X10”, where a node failure is reported as a `DeadPlaceException` and the execution can continue using the remaining nodes. In this paper, we explain how to construct fault-tolerant distributed applications using Resilient X10 functions. Three basic methods are introduced to handle the node failures: (a) *Ignore* failures and use the results from the remaining nodes, (b) *Reassign* the failed node’s work to the remaining nodes, or (c) *Restore* the computation from a periodic snapshot. We also describe a fault-tolerant extension of the existing distributed X10 library `DistArray`. These modifications to add fault tolerance were very small, and the modified code can still run on standard X10 as long as node failure does not occur. The impacts on execution performance caused by the modifications are also evaluated.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—distributed programming

General Terms Reliability, Algorithms, Language, Design, Experimentation

Keywords Resilient X10, Fault tolerance, Distributed programming

1. Introduction

X10 [25] is a programming language that supports distributed computing. However in the original X10 implementation, when a computing node crashes, the whole X10 application is aborted even if the other nodes are alive. This may become a problem especially in exascale computing where a massive number of nodes are used for the computation. For example, there is a report that MTBF (Mean Time Between Failures) for 1,408 compute nodes was 15.8 hours [18]. To address this problem, we are developing an extended version of X10 called “Resilient X10” [3], where the node failure is reported as an exception and the execution can continue using the remaining nodes. This paper explains how fault-tolerant distributed applications can be constructed using Resilient X10 functions.

For the execution model on a distributed environment, X10 uses PGAS (Partitioned Global Address Space) [15]. As the name indicates, the PGAS model provides a global address space, but it is partitioned into multiple *places*, each of which basically corresponds to a computing node. Data can be referenced from any place, but an activity must move to the place using an “`at`” statement to access the data.

A node failure can be considered as the death of a place in X10, which means that data on the place become inaccessible. However, since the address space is explicitly partitioned, it is relatively easy in the PGAS model to continue the execution by *detaching* the failed node (place). For example, when the target place of an `at` statement dies, the processing can continue if the caller receives some kind of notification. In the Resilient X10 we are developing, a newly defined exception, `DeadPlaceException`, is raised for this situation.

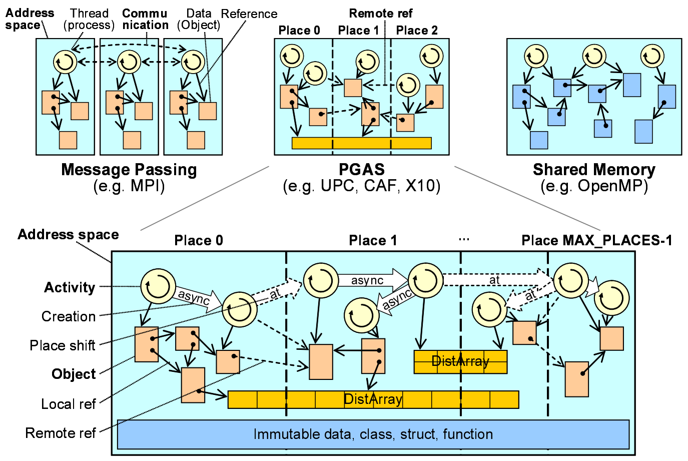

Figure 1. Execution model of X10.

Unfortunately, simple notification of the place death is not sufficient to continue the application. To achieve fault tolerance, the application (or libraries) must reconfigure the processing appropriately based on the notification. In this paper, we introduce the following three basic methods to handle the notification: (a) Ignore failures and use the results from the remaining nodes, (b) Reassign the failed node’s work to the remaining nodes, or (c) Restore the computation from a periodic snapshot. Programming examples for these three methods in Resilient X10 are provided. We also show a fault-tolerant extension of the existing distributed X10 library `DistArray`. These modifications to add fault tolerance are very small, and the modified code can still run on standard X10 as long as node failure does not occur. The major contributions of this paper are:

- An analysis of multiple approaches to add fault tolerance to existing distributed applications, proving that they can be implemented with minimal modifications using Resilient X10.
- An evaluation of the developed fault-tolerant applications from various perspectives in a real distributed environment.

2. Fault Tolerance Support in Resilient X10

For programming parallel and distributed applications, it is very important how the parallelism and distributed memory structure are exposed to the programmer. The upper figures in Figure 1 show three variations. In a “shared memory” model such as OpenMP [14], an address space is shared by all of the nodes (cores), and explicit data division or communication is unnecessary. This simplifies the programming, but fine-grain tuning to maximize the use of the hardware may be difficult. In contrast, a “message passing” model such as MPI [11], where all of the communications are explicitly described, always requires the programmer to be fully aware of the underlying hardware configuration.

The PGAS used by X10 can be regarded as being between these two approaches, with a global address space that is partitioned into

```
1  class HelloWorld {
2    public static def main(args:Rail[String]) {
3      finish for (pl in Place.places()) {
4        at (pl) async { // parallel distributed exec in each place
5          Console.OUT.println("Hello from " + here);
6        }
7      } // end of finish, wait for the execution in all places
8    }
9  }
10     $ x10c++ HelloWorld.x10 -o HelloWorld # compile
11     $ X10_NPLACES=4 runx10 HelloWorld     # execute
12     Hello from Place(3)
13     Hello from Place(0)
14     Hello from Place(2)
15     Hello from Place(1)
```

Figure 2. Parallel distributed Hello World in X10.

multiple "places". The place is an abstraction of memory locality and typically corresponds to a computing node. Each unit of data belongs to a specific place and can be accessed only in that place, but it can be referenced (pointed to) from any place. By providing such an abstracted view of the underlying parallel/distributed environment instead of concealing it, X10 tries to offer high productivity for high performance computing. The PGAS model is also used in other languages such as UPC (Unified Parallel C) [21], CAF (Co-Array Fortran) [23], and XMP (XcalableMP) [26]. Since X10 supports the arbitrary creation of asynchronous activities, its execution model is sometimes called APGAS (Asynchronous PGAS).

2.1 APGAS Execution Model

The lower part of Figure 1 shows X10's APGAS execution model. A global address space is partitioned into multiple *places*, which can hold multiple *activities* and *objects*.

An activity is a kind of lightweight thread sequentially executed in a place. It is created by an `async` statement and can move to another place using an `at` statement, through which parallel and distributed processing is made possible. A `finish` statement is used to wait for the termination of activities (including grandchildren) created inside that block. At this time, exceptions are propagated by being encapsulated in a `MultipleExceptions` object. By using this "Rooted Exception Model", X10 programs can handle exceptions thrown by asynchronous activities.

An object is a mutable data structure that belongs to a specific place and can only be accessed by the activities running in that place. However, objects can be globally (remotely) referenced by using `GlobalRef` [7, 17]. The global reference contains information about where the object exists, so activities can move to that place to access the object. For example, a statement "`data = at (gref) gref();`" indicates to move to the place of the global reference `gref` and read its value. X10 also provides a distributed array structure named `DistArray` whose elements are scattered over multiple places. Each element can be manipulated by activities in the same place.

Figure 2 shows a Hello World program written in X10 and an execution example using four places. A new asynchronous activity is created in each of the places by the `at` and `async` statements in Line 4, and a message "`Hello from Place(`*n*`)`" is printed in a parallel and distributed manner. The termination of the activities are waited for by the `finish` statement ending at Line 7, and the program terminates its execution.

2.2 Resilient X10

Distributed processing with X10 can be done by running each place as a process on a different computing node. When a node is broken, all of the activities and data in that place are lost, and the entire X10 processing is aborted in standard X10. However, since the address space partitioning is visible from applications, localizing the impact of place death is relatively easy in the PGAS model. Consider the case when Place 1 in the lower part of Figure 1 dies. Three activities, two objects, and some parts of two distributed arrays in that place are lost, but the remaining portions in other places are not affected. Global references to the dead place became inaccessible, but can remain in live places. By exploiting this, we have been extending X10 to continue the execution using the remaining nodes (places), in our new language "Resilient X10" [3].

```
1  class ResilientExample {
2    public static def main(Rail[String]) {
3      finish for (pl in Place.places()) async {
4        try {
5          at (pl) do_something(); // parallel distributed execution
6        } catch (e:DeadPlaceException) {
7          Console.OUT.println(e.place + " died"); // report failure
8        }
9      } // end of finish, wait for the execution in all places
10   }
11 }
```

Figure 3. A simple fault-tolerant program which just reports node failures.

In Resilient X10, the newly defined exception `DeadPlaceException` (shortened to `DPE` in the following explanation) is thrown for a place death. Concretely, if an activity is being moved (or trying to move) to a dead place, the corresponding `at` statement throws the `DPE`. When an activity is asynchronously executing in a dead place using the `async` statement, the `finish` statement governing the activity will throw a `MultipleExceptions` object which contains the `DPE` (and possibly other exceptions in the block). The ID of the dead place can be checked through the `place` field of the `DPE`. Some utility methods are also provided, such as `Place.isDead` to check if a specified place is dead or not, and `Place.numDead` to check the number of dead places[1].

These functions make it possible to write fault-tolerant X10 applications that can continue to run on the remaining nodes even when some places are lost due to node failures. Figure 3 shows a simple example. This program performs `do_something` in all of the places, and simply prints "`Place(`*n*`) died`" for the `DPE` notification (Lines 6–7).

To process a `finish` statement appropriately even when some activities have been lost due to node failure, the execution status of the activities (the number of activities running on each place) governed by the `finish` must be recorded securely and not lost by the failure. Resilient X10 stores such kinds of critical information in a reliable storage area named *Resilient Storage*. We have implemented several variations of the Resilient Storage, and the most stable version in our current implementation is the one which uses Place 0 (the place where `main` is executed) for this purpose[2].

The Resilient X10 function is included as a technology preview in X10 2.4.1 (released in December 2013) and later versions, and can be enabled by specifying an environment "`X10_RESILIENT_MODE=1`". Resilient X10 can run with either of two X10 implementations: Native X10, compiled to C++ and executed natively, and Managed X10 [7, 20], compiled to Java^{TM} and executed on multiple Java VMs. The communication layer is limited to sockets, and MPI [11] and PAMI [8] are not yet supported. Refer to [3] for more implementation details, and [2] for the semantics of the place-death handling.

[1] In the current implementation, the place IDs are not reassigned when a place has died. The `Place.MAX_PLACES` field always holds the number of places at the start-up time, and the `Place.places` method returns a list of places including any dead places. These constraints are mainly for the backward compatibility, but may be changed in the future when dynamic place addition is supported.

[2] This version has a limitation that the X10 execution is aborted if Place 0 dies. The fault-tolerant applications shown in this paper also use the assumption that Place 0 never dies. Note that even with this limitation, the MTBF of the system is greatly extended, becoming to the MTBF of a single node (Place 0).

```
 1 import x10.util.*;
 2 class ResilientMontePi {
 3   static val ITERS = 1000000000/Place.MAX_PLACES; // one billion
 4   public static def main (args:Rail[String]) {
 5     val result = GlobalRef(new Cell(Pair[Long,Long](0,0)));
 6         // global result which holds (#inside the circle, #trials)
 7     finish for (p in Place.places()) async {
 8       try {
 9         at (p) {
10           val rnd = new x10.util.Random(System.nanoTime());
11           var c:Long = 0;
12           for (iter in 1..ITERS) { // ITERS trials per place
13             val x = rnd.nextDouble(), y = rnd.nextDouble();
14             if (x*x + y*y <= 1.0) c++; // if inside the circle
15           }
16           val count = c;
17           at (result) atomic { // update the global result
18             val r = result();
19             r() = Pair(r().first+count, r().second+ITERS);
20           }
21         }
22       } catch (e:DeadPlaceException) {
23         // just ignore the results of dead places
24       }
25     } // end of finish, wait for the execution in all places
26     val pair = result()();
27     val pi = 4.0 * pair.first / pair.second;
28     Console.OUT.println("pi="+pi + " (try="+pair.second+")");
29   }
30 }
```

Figure 4. Computation of π with the Monte Carlo method.

```
 1 class ResilientKMeans {
 2   static val POINTS = 10000000; // number of points
 3         :
 4   public static def main(args:Rail[String]) {
 5     /* prepare a set of points, and deliver it to other places */
 6     for (iter in 1..ITERATIONS) { // iterate until convergence
 7       /* deliver current cluster values to other places */
 8       // process some part of the points at each place
 9       val numAvail = Place.MAX_PLACES - Place.numDead();
10       val div = POINTS / numAvail; // share for each place
11       val rem = POINTS % numAvail; // extra share for Place 0
12       var start:Long = 0; // next point to be processed
13       try {
14         finish for (pl in Place.places()) {
15           if (pl.isDead()) continue; // skip dead place(s)
16           var end:Long = start+div; if (pl==place0) end+=rem;
17           at (pl) async { // compute at live places in parallel
18             /* process points [start,end), and return the data
19                necessary for updating cluster vals to Place 0 */
20           }
21           start = end;
22         } // end of finish, wait for the execution in all places
23       } catch (es:MultipleExceptions) {
24         for (e in es.exceptions()) { // just ignore place death
25           if (!(e instanceof DeadPlaceException)) throw e; }
26       }
27       /* compute new cluster values, and exit if converged */
28     } // end of for (iter)
29     /* print the result */
30   } // end of main
31 }
```

Figure 5. Fault-tolerant KMeans (skeleton).

3. Fault-Tolerant X10 Applications

Currently, Resilient X10 provides only a few programming interfaces: `DeadPlaceException` to notify the application of place death and support methods such as `Place.isDead` and `Place.numDead`, but these are sufficient to add fault tolerance to existing distributed X10 applications.

However for this to work, it is necessary to understand how the application is doing the distributed processing and how the execution can be continued after a node failure. Depending on these situations, the approach to adding the fault tolerance will differ. This section explains two methods to handle node failures:

(a) Ignore the failures and use the results from the remaining nodes,

(b) Reassign the failed node's work to the remaining nodes.

A third method will be added in Section 4.

3.1 MontePi

Figure 4 is an example of adding fault tolerance to an application which approximates π by using a Monte Carlo method. The core part of the computation is the `for` loop at Lines 12–15, which repeatedly checks if randomly generated coordinates (`x`, `y`) are inside a circle of radius 1.0. After `ITERS` number of trials, the result is added to a global `result` data pair stored at Place 0 (Lines 17–20). This series of processes is performed in parallel at each place, and when all of them finish, the value of π is finally calculated from the `result` (Lines 26–27).

If a node fails during the computation, the place executing on that node dies and the `at` statement at Line 9 throws a `DeadPlaceException`. This exception is caught by the `catch` statement at Line 22 and just ignored. In the now dead place, the `at` statement at Line 17 is never executed, so the global `result` is not updated[3]. Therefore, only the results from living nodes will be used for the final calculation. The result may become less accurate because the number of trials of the Monte Carlo method is reduced, but it is not completely broken by the node failure. This program also prints the total number of trials as reference information.

In this example, the code that was added for fault tolerance is only the `try` statement at Line 8 and the `catch` statement spanning Lines 22–24.

3.2 KMeans

KMeans is a basic algorithm for clustering analysis, categorizing n points in d-dimensional space into K clusters[4]. The basic computation is an iteration of: (1) Categorizing the n points based on the coordinates of the K clusters, and (2) Making the centroid of the points to the new coordinates of each cluster. The iteration ends when the differences in the coordinates between iterations become smaller than a threshold. In a distributed environment, Step (1) can be done in parallel by assigning n points to each computing node and returning the results necessary for Step (2). Since the coordinates of the n points do not change, they can be copied in advance.

For such programs, fault tolerance can be supported by *reassigning* the work only to available nodes. Figure 5 shows the skeleton of a fault-tolerant KMeans application based on this approach. In this figure, "`/*` $\cdots$ `*/`" explains the processing of the omitted part. The complete code is listed in Figure 9 of the appendix. The work reassignment is performed by the code in Lines 9–17. The number of available places is determined in Line 9, and the number of points to be processed at each place is calculated in Lines 10–11. Lines 14–17 process the computation at the live places in parallel by skipping the dead places (Line 15).

When a place dies during the execution, a `DeadPlaceException` (enclosed by a `MultipleExceptions`) is thrown for the `finish` statement in Line 14, and caught by the `catch` statement in Line 23. The notification is simply ignored in this program, and retry of the lost computation or disposal of other results are not performed. This is because even the partial results can still make the convergence faster for the KMeans computation (we call this approach *decimation* [3]). However, to make the final results precise, the convergence check is performed only when all of the points are processed.

[3] If the place dies after executing the `at` statement at Line 17, the result will be used for the final calculation. The body of this `at` is executed at Place 0, so the result is updated correctly even if the originating place died.

[4] In the example in this paper, $n = 10,000,000$, $d = 4$, and $K = 4$.

```
 1 package x10.regionarray;
 2 public class ResilientDistArray[T] ... {
 3   public static def make[T](dist:Dist, init:(Point)=>T)
 4                                     : ResilientDistArray[T];
 5   public static def make[T](dist:Dist){T haszero}
 6                                     : ResilientDistArray[T];
 7   public final operator this(pt:Point) : T; // read element
 8   public final operator this(pt:Point)=(v:T) : T; // set element
 9   public final def map[S,U](dst:ResilientDistArray[S],
10        src:ResilientDistArray[U], filter:Region, op:(T,U)=>S)
11                                     : ResilientDistArray[S];
12   public final def reduce(op:(T,T)=>T, unit:T) : T;
13          :
14   // Create a snapshot
15   public def snapshot() { snapshot_try(); snapshot_commit(); }
16   public def snapshot_try() : void; // may throw DPE
17   public def snapshot_commit() : void;
18   // Reconstruct the DistArray with new Dist
19   public def restore(newDist:Dist) : void;
20   public def remake(newDist:Dist, init:(Point)=>T) : void;
21   public def remake(newDist:Dist){T haszero} : void;
22 }
```

Figure 6. Interface of the DistArray with snapshot mechanisms.

In this example, the code added for fault tolerance is about 10 lines, including the parts omitted from Figure 5.

4. Fault-Tolerant X10 Libraries

The MontePi and KMeans examples in the previous section were written only using basic X10 constructs. However, for efficiently writing large distributed applications, library support is crucial. As one of these libraries, X10 supports distributed array (`DistArray`), where the elements are scattered over multiple places, similar to the Co-Array in CAF. This section introduces an extension named "Resilient DistArray", which supports snapshot mechanisms to handle node failures, and a fault-tolerant application using the mechanism. This leads to a third method to handle node failures:

(c) Restore the computation from a periodic snapshot.

4.1 Resilient DistArray

A large-scale distributed application typically has an SPMD structure, where each computing node executes the same processing on a small part of the large amount of data. A `DistArray` is a data structure suitable for such processing. The elements of the array are divided into multiple places and processed by activities running on the owning places. How to distribute the elements among the places can be flexibly defined by using a `Dist` structure. See the X10 documents [17] for the details.

Upon a node failure, the `DistArray` elements belonging to the dead place become inaccessible, so the SPMD processing cannot continue. To resume the processing, the distributed array must be rearranged among the remaining places while restoring all of the element data. The `ResilientDistArray` is a fault-tolerant extension[5] of the distributed array to support this function. The current version is implemented as an independent class which contains a `DistArray` field, requiring about 200 lines of code. We are considering integrating the function into the standard distributed array in the future.

Figure 6 shows the interface of the `ResilientDistArray`. The `[T]` is a type parameter, `(T)=>U` is a function type, and the right part of colon specifies the type of the argument or return value. As in standard `DistArray`, it can be created by the `make` method with a `Dist` argument to specify the distribution of the elements and an optional function argument for initialization. Each element can be accessed in the owning place by using the form "`A(pt)`". Utility methods like `map` and `reduce` are also available.

[5] Currently X10 has two implementations of distributed array [17]. The one in `x10.regionarray` package supports complex and flexible element distribution, and the other in `x10.array` package is simple, but faster. There is a fault-tolerant extension for both of them, but the explanation in this paper uses the `x10.regionarray` implementation.

```
 1 class ResilientHeatTransfer {
 2   static val N = 20; // size of grid
 3   static val livePlaces = new ArrayList[Place]();
 4   static val restore_needed = new Cell[Boolean](false);
 5           :
 6   public static def main(args:Rail[String]) {
 7     for (pl in Place.places()) livePlaces.add(pl);
 8     // initialize Region and Dist
 9     val BigR = Region.make(0..(N+1), 0..(N+1)); // +surroundings
10     var BigD:Dist(2) = Dist.makeBlock(BigR, 0,
11                 new SparsePlaceGroup(livePlaces.toRail()));
12     // create a DistArray, each element holds a heat value
13     val A = ResilientDistArray.make[Double](BigD, ...);
14     A.snapshot(); // create the initial snapshot
15     for (iter in 1..ITERATIONS) { // iterate until convergence
16       try {
17         if (restore_needed()) { // if some places died
18           BigD = Dist.makeBlock(BigR, 0, // recreate Dist, and
19                 new SparsePlaceGroup(livePlaces.toRail()));
20           A.restore(BigD); // restore elements from the snapshot
21           restore_needed() = false;
22         }
23         finish ateach (z in D_Base) { // distributed processing
24           /* compute new heat values for A's local elements */
25         }
26         /* if converged, exit the for loop */
27         if (iter % 10 == 0) A.snapshot(); // create a snapshot
28       } catch (e:Exception) { processException(e); }
29     } // end of for (iter)
30     /* print the result */
31   } // end of main
32 
33   private static def processException(e:Exception) { // exception
34     if (e instanceof DeadPlaceException) {
35       val deadPlace = (e as DeadPlaceException).place;
36       livePlaces.remove(deadPlace); restore_needed() = true;
37     } else ... /* handle MultiPlaceExceptions recursively */
38   }
39 }
```

Figure 7. Fault-tolerant HeatTransfer (skeleton).

The fault-tolerant extension starts from Line 14. The `snapshot` method dumps the element values into the Resilient Storage, which can be restored by the `restore` method. At this time, the elements can be rearranged among the live places based on the `newDist` argument. The `remake` method can be used to re-initialize the array with the new distribution without restoring any data. Using these functions to create periodic snapshots of intermediate data, it is possible to restart the computation from the snapshot by reconstructing the DistArrays over the remaining live places.

4.2 HeatTransfer

As an example using the `ResilientDistArray`, we selected the "HeatTransfer" program in this section. This is an application to compute the heat diffusion through a two-dimensional grid represented by an array. Each element of the array holds the heat value of a grid point, and is repeatedly updated by the average of surrounding four points until convergence. This is a "stencil" computation pattern, which is very common in HPC applications. By using `DistArray`, HeatTransfer can be easily implemented as a distributed program using multiple computing nodes, and several variations are included as examples in the X10 distribution.

Figure 7 shows the skeleton of a fault-tolerant HeatTransfer application using `ResilientDistArray`. The complete code is listed in Figure 10 of the appendix. The `for` loop of Lines 15–29 is repeated until the result converges. Inside the loop, for each element of the DistArray, a new heat value is calculated in parallel at each place (Lines 23–25). To support fault tolerance, a new snapshot of DistArray `A` is created at every 10th iteration (Line 27).

When a node failure occurs during the execution, a `DeadPlaceException` (enclosed by a `MultipleExceptions`) is caught by the `catch` statement in Line 28, and the `processException` method is called. This method removes the dead place from the `livePlaces` list and sets the `restore_needed` flag (Line 36). If this flag is set at the beginning of an iteration (Line 17), then DistArray `A` is reconstructed over the remaining places using the `livePlaces` information, and the element values are restored from the latest snapshot (Lines 18–20). The execution is slightly unwound by this, but can be resumed by detaching the failed node.

In this example, the modifications for fault tolerance other than replacing `DistArray` with `ResilientDistArray` involve about 25 lines, including the parts omitted from Figure 7. This may seem to be relatively large for such a small application, but we believe larger DistArray applications can be made fault tolerant using the same approach, and the modification ratio will be smaller for more typical applications. Since half of the modifications are in the exception-handling code of `processException`, it can also be considered to provide this method as a library.

5. Evaluation

This section shows various evaluations of fault tolerance support using the applications from previous sections: MontePi, KMeans, and HeatTransfer.

5.1 Modification Amount

As explained in each section, the modifications necessary to add fault tolerance were very small: only 4 lines for MontePi, about 10 lines for KMeans, and about 25 lines for HeatTransfer. Adding the snapshot and restore functions to DistArray was done in about 200 lines of code. Note that all of these fault-tolerant programs will still run on standard X10 as long as node failure does not occur.

As shown above, Resilient X10 makes it possible to add fault tolerance to existing distributed applications with very small modifications. However, the best approach to adding the fault tolerance depends on the structure of each application. Therefore, we discussed three typical methods in this paper: (a) Ignore failures and use the results from the remaining nodes (MontePi), (b) Reassign the failed node's work to the remaining nodes (KMeans), or (c) Restore the computation from a periodic snapshot (HeatTransfer).

5.2 Execution Performance

Next, we evaluated the impact of fault tolerance support on the execution performance. For each application, its base code and a fault-tolerant version were executed on standard X10 and Resilient X10, and the computation times were compared. All of the measurements were done using four IBM® BladeCenter® HS23 (7875-C5J) blades, each of which consists of two 2.7-GHz Intel® Xeon® E5-2680 processors (a total of 16 cores). The machines were interconnected with 40-Gbps InfiniBand and running Red Hat Enterprise Linux® Server 6.3. The X10 was the Native X10 2.4.2 using the sockets communication layer, and Place 0 was used as the Resilient Storage. Each machine ran two places, so a total 8 places were used for the execution. Each application was compiled with "`x10c++ -O -NO_CHECKS`", and the time for executing the outermost `for` loop was measured. The print statements inside the loop were disabled. Each measurement was done 10 times and the best score was used.

Figure 8 shows the relative execution times normalized by the time of base code on standard X10 for each application. From left to right, each bar shows the score of the base code and its fault-tolerant version on standard X10, and their times on Resilient X10, respectively.

Comparing the execution times on standard X10, almost no overheads due to adding fault tolerance were observed for MontePi and KMeans. For HeatTransfer, the fault-tolerant version took 9% more time, because of the cost of the periodic snapshots.

Next, the performance of each base code on standard and Resilient X10 was compared. In Resilient X10, the costs of `at` and

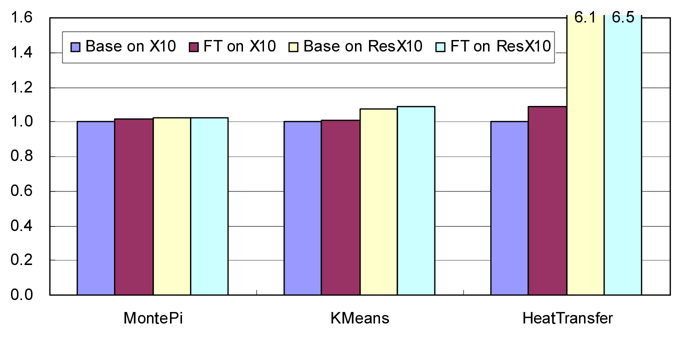

Figure 8. Relative execution times.

`async` are larger since the critical states for the `finish` handling must be recorded in the Resilient Storage. Therefore, the execution becomes slightly slower, even for the base code. MontePi became about 2% slower because two `at` statements are executed for each place. KMeans became 8% slower since `at` is performed multiple times until the convergence. For HeatTransfer, the execution time increased by 6 times. This is mainly because `at` is invoked too frequently in the internal stencil computation.

By combining these overheads, compared to the case where the base code is executed on standard X10, running fault-tolerant code on Resilient X10 had 2.2% more overhead for MontePi and 9.0% more for KMeans. In contrast, HeatTransfer suffered a 6.5-fold slowdown. This can change with the snapshot frequency, and may be reduced by rewriting the application not to use unnecessary `at` calls. Adding "halo region" support to DistArray [10] can also reduce the overhead.

5.3 Fault Tolerance

All of the results shown in the previous section are cases where no node failures occurred. Finally, we studied the behavior when nodes failed.

When an X10 process which corresponds to Place 2 was terminated externally by a `kill` command, the entire processing was aborted on standard X10 regardless of the fault tolerance of the application. Even in Resilient X10, base code applications were terminated by the `DeadPlaceException` (or enclosing `MultipleExceptions`). In contrast, when fault-tolerant versions were executed on Resilient X10, they could survive the place death and output the correct results. This means that the combination of fault-tolerant applications and Resilient X10 provided good fault tolerance. An execution example for HeatTransfer with place deaths is included in the research report version of this paper [6].

The effects caused by place deaths were also measured by running the fault-tolerant applications on Resilient X10. In MontePi, the deaths may reduce the accuracy of the results since the number of trials decreases. When 4 places among the 8 places were killed, the execution time did not increase, but the deviation of the calculated π value increased from 0.0008% to 0.002%. In KMeans and HeatTransfer, the place death increases both the number of iterations until convergence and the execution time of each iteration. When Place 2 was killed during the execution of the 17th iteration, execution time increased by 11% in KMeans and 14% in HeatTransfer, but the executions still ended with correct results.

6. Related Work

Usually, distributed processing is supported through libraries such as MPI [11], RMI [5], and the DB/Web access packages. In such cases, a node failure appears as a low-level error in the communication code. Therefore, fault tolerance must be supported by each application as error handling for the communication routines. In contrast, Resilient X10 can handle the node failure as part of its computation model [2, 3] by utilizing the characteristics of PGAS. Application modifications are still necessary to support fault tolerance, but we believe most cases are covered by the patterns presented in this paper. In addition, fault tolerance can be built into the application from the beginning, because the support exists inside the language.

How to handle node failures has been an important issue, especially in HPC applications that use massive numbers of computing nodes. The most popular approach is *checkpointing* [4, 18] implemented by each application. Intermediate data is periodically saved, and when a failure occurs, the restarted application resumes the processing by restoring the data from the snapshot. The fault-tolerant HeatTransfer in Section 4.2 is based on the same idea, but most of the save and restore mechanisms are implemented in the general-purpose DistArray library, which makes the mechanisms easier to use. In addition, the application does not need to be restarted after node failures. In Hadoop [24], fault tolerance is achieved by writing the results of each phase to external storage (HDFS). This can be considered as a variation of the checkpointing approach.

Instead of restoring the data to get precise results, there is an interesting fault-tolerance approach of *discarding* only the failed results. The MontePi and KMeans in this paper can be considered as simple examples of this approach, but more detailed modeling to calculate *probabilistic accuracy bounds* has also been proposed [16]. We will take such models into account in developing larger fault-tolerant applications.

Recently, it has become common to perform large scale computations on virtualized (cloud) environments. In such situations, fault tolerance may be achieved by utilizing the snapshot [22] or migration [1, 9] functions for virtual machines [13]. However, this creates overhead to save or move entire virtual machines, so additional research is necessary to utilize them for distributed applications that use multiple computing nodes. One of the interesting topics is how Resilient X10 and its applications can benefit from virtualization.

7. Conclusions and Future Work

This paper described how fault-tolerant applications can be constructed using the “Resilient X10” extensions of the distributed programming language X10. Three methods of adding fault tolerance to existing applications are shown: (a) *Ignore* failures and use the results from the remaining nodes, (b) *Reassign* the failed node’s work to the remaining nodes, or (c) *Restore* the computation from a periodic snapshot. For all of these approaches, the code modifications were less than 25 lines. We also showed a distributed library, Resilient DistArray, which supports snapshot mechanisms.

Compared to the case where the original application is executed on standard X10, running fault-tolerant applications on Resilient X10 has 2.2% to 9.0% overhead if place change does not happen too frequently. We also noticed that there is a pathological case with a 6.5-fold slowdown if place changes are performed too frequently. The major cause of this overhead was the cost of periodic snapshots and the additional costs of `at` processing. However, by paying these costs, we confirmed that fault-tolerant applications can survive node failures. When one place among 8 places was lost, the execution time increased by 11–14%, although the exact number depends on the timing of the place death.

In the future, we want to reduce this overhead in fault-tolerant applications by improving both the applications and Resilient X10. Preparing more libraries that support fault tolerance is another topic. One candidate should be an interface to access the Resilient Storage from applications. We also plan to make fault-tolerant versions of larger distributed X10 applications, such as those described in [12] and [19]. Removing the dependency on Place 0 is also an important topic. We already have a prototype for distributed Resilient Storage, which allows the `finish` states and DistArray snapshots to be stored in a distributed manner, rather than in Place 0.

Acknowledgments

We would like to thank the members of X10 project in IBM T. J. Watson Research Center and IBM Research - Tokyo, for their various suggestions and valuable comments. The Resilient X10 research was funded in part by the U. S. Air Force Office of Scientific Research under Contract No. FA8750-13-C-0052.

References

[1] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI ’05)*, pages 273–286, 2005.

[2] S. Crafa, D. Cunningham, V. Saraswat, A. Shinnar, and O. Tardieu. Semantics of (Resilient) X10. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP ’14)*, 2014, to appear.

[3] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu. Resilient X10: Efficient Failure-Aware Programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’14)*, pages 67–80, 2014.

[4] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Survey*, 34(3):375–408, 2002.

[5] W. Grosso. *Java RMI*. O’Reilly, 2001.

[6] K. Kawachiya. Writing Fault-Tolerant Applications Using Resilient X10. Research Report RT0960, IBM Research - Tokyo, 2014.

[7] K. Kawachiya, M. Takeuchi, S. Zakirov, and T. Onodera. Distributed Garbage Collection for Managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop (X10 ’12)*, 2012.

[8] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS ’12)*, 2012.

[9] V. Medina and J. M. Garcia. A Survey of Migration Mechanisms of Virtual Machines. *ACM Computing Survey*, 46(3), 2014.

[10] J. Milthorpe and A. P. Rendell. Efficient Update of Ghost Regions Using Active Messages. In *Proceedings of the 2012 Annual IEEE International Conference on High Performance Computing (HiPC ’12)*, 2012.

[11] MPI Forum. Message Passing Interface Forum. `http://www.mpi-forum.org/`.

[12] H. Murata, M. Horie, K. Shirahata, J. Doi, H. Tai, M. Takeuchi, and K. Kawachiya. Porting MPI based HPC Applications to X10. In *Proceedings of the 2014 X10 Workshop (X10 ’14)*, 2014.

[13] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS ’07)*, pages 23–32, 2012.

[14] OpenMP.org. The OpenMP: API Specification for Parallel Programming. `http://www.openmp.org/`.

[15] PGAS. Partitioned Global Address Space. `http://www.pgas.org/`.

[16] M. Rinard. Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS ’06)*, pages 324–334, 2006.

[17] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[18] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and Modeling of a Non-blocking Checkpointing System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2012 (SC ’12)*, 2012.

[19] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased Performance for In-Memory Hadoop Jobs. *Proceedings of the VLDB Endowment*, 5 (12):1736–1747, 2012.

[20] 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 (X10 ’11)*, 2011.

[21] The High Performance Computing Laboratory. Unified Parallel C at George Washington University. `http://upc.gwu.edu/`.

[22] VMware Knowledge Base. Understanding Virtual Machine Snapshots in VMware ESXi and ESX. `http://kb.vmware.com/selfservice/microsites/search.do?cmd=displayKC&externalId=1015180`.

[23] A. Wallcraft. Co-Array Fortran Homepage. `http://www.co-array.org/`.

[24] T. White. *Hadoop: The Definitive Guide, 3rd Edition*. O’Reilly Media / Yahoo Press, 2012.

[25] X10 Project. X10: Performance and Productivity at Scale. `http://x10-lang.org/`.

[26] XcalableMP Project. XcalableMP: Directive-based language eXtension for Scalable and performance-aware Parallel Programming. `http://www.xcalablemp.org/`.

```
import x10.regionarray.*;
class ResilientKMeans {
  static val DIM = 4n;          // number of dimensions
  static val POINTS = 10000000; // number of points
  static val CLUSTERS = 4;      // number of clusters to be categorized
  static val ITERATIONS = 1000; // number of maximum iterations

  public static def main(args:Rail[String]) {
    val place0 = here;
    // prepare a set of points (coordinates of i-th point are [pt(i,0),pt(i,1),pt(i,2),pt(i,3)]), which do not change after prepared
    val points_region = Region.make(0..(POINTS-1), 0..(DIM-1)), rnd = new x10.util.Random(0);
    val points_master = new Array[Float](points_region, (p:Point)=>rnd.nextFloat());
    val points_local =
    PlaceLocalHandle.make[Array[Float]](PlaceGroup.WORLD, ()=>points_master); // deliver the point set to other places
    // an array to hold the cluster values (coordinates of k-th cluster are [cl(k*4),cl(k*4+1),cl(k*4+2),cl(k*4+3)])
    val central_clusters = new Rail[Float](CLUSTERS*DIM, (i:Long)=>points_master(i/DIM, i%DIM)); // use i-th point as initial value

    // prepare data structures for the computation
    val old_central_clusters = new Rail[Float](CLUSTERS*DIM); // an array to hold the previous cluster values
    val central_cluster_counts = new Rail[Long](CLUSTERS); // number of points in each cluster
    val processed_points = new Cell[Long](0); // number of processed points
    // prepare global refs for remote access
    val central_clusters_gr = GlobalRef(central_clusters);
    val central_cluster_counts_gr = GlobalRef(central_cluster_counts);
    val processed_points_gr = GlobalRef(processed_points);
    // prepare three local arrays for processing at each place
    val local_curr_clusters = PlaceLocalHandle.make[Rail[Float]](PlaceGroup.WORLD, ()=>new Rail[Float](CLUSTERS*DIM));
    val local_new_clusters  = PlaceLocalHandle.make[Rail[Float]](PlaceGroup.WORLD, ()=>new Rail[Float](CLUSTERS*DIM));
    val local_cluster_counts = PlaceLocalHandle.make[Rail[Long]](PlaceGroup.WORLD, ()=>new Rail[Long](CLUSTERS));

    for (iter in 1..ITERATIONS) { Console.OUT.println("Iteration " + iter); // iterate until the result converges
      // 1. deliver current cluster values to other places
      try {
        finish for (pl in Place.places()) { if (pl.isDead()) continue; // skip dead place(s)
          at (pl) async { // compute at live places in parallel
            for (var j:Long = 0; j < CLUSTERS*DIM; ++j) {
              local_curr_clusters()(j) = central_clusters(j); local_new_clusters()(j) = 0f; }
            for (var j:Long = 0; j < CLUSTERS; ++j) local_cluster_counts()(j) = 0;
        } }
      } catch (es:MultipleExceptions) {
        for (e in es.exceptions()) { if (!(e instanceof DeadPlaceException)) throw e; } // just ignore place death
      }
      // 2. save current cluster values and clear them
      for (var j:Long = 0; j < CLUSTERS*DIM; ++j) { old_central_clusters(j) = central_clusters(j); central_clusters(j) = 0f; }
      for (var j:Long = 0; j < CLUSTERS; ++j) central_cluster_counts(j) = 0; processed_points() = 0;
      // 3. process some part of the points at each place
      val numAvail = Place.MAX_PLACES - Place.numDead(); // number of live places
      val div = POINTS / numAvail, rem = POINTS % numAvail; // share for each place, and extra share for Place 0
      var start:Long = 0; // next point to be processed
      try {
        finish for (pl in Place.places()) { if (pl.isDead()) continue; // skip dead place(s)
          var end:Long = start + div; if (pl==place0) end += rem; // points [start,end) are processed in this place
          val s = start, e = end;
          at (pl) async { // compute at live places in parallel
            for (var j:Long = s; j < e; ++j) { val p = j; // process the p-th point
              val points = points_local(); var closest:Long = -1, closest_dist:Float = Float.MAX_VALUE;
              for (var k:Long = 0; k < CLUSTERS; ++k) { // find the closest cluster
                var dist:Float = 0f;
                for (var d:Long = 0; d < DIM; ++d) { // calculate the distance to the k-th cluster
                  val tmp = points(p,d) - local_curr_clusters()(k*DIM+d); dist += tmp * tmp; }
                if (dist < closest_dist) { closest_dist = dist; closest = k; }
              }
              local_cluster_counts()(closest)++; // add the coordinates of the point to the closest cluster
              for (var d:Long = 0; d < DIM; ++d) local_new_clusters()(closest*DIM+d) += points(p,d);
            } // end of the processing of assigned points
            val tmp_new_clusters = local_new_clusters(), tmp_cluster_counts = local_cluster_counts(), tmp_processed_points = e-s;
            at (place0) atomic { // return the results to the master
              for (var j:Long = 0; j < CLUSTERS*DIM; ++j) central_clusters_gr()(j) += tmp_new_clusters(j);
              for (var j:Long = 0; j < CLUSTERS; ++j) central_cluster_counts_gr()(j) += tmp_cluster_counts(j);
              processed_points_gr()() += tmp_processed_points;
          } }
          start = end;
        } // end of finish, wait for the execution in all places
      } catch (es:MultipleExceptions) {
        for (e in es.exceptions()) { if (!(e instanceof DeadPlaceException)) throw e; } // just ignore place death
      }
      // 4. compute new cluster values, and check the convergence
      for (var k:Long = 0; k < CLUSTERS; ++k)
        for (var d:Long = 0; d < DIM; ++d) central_clusters(k*DIM+d) /= central_cluster_counts(k);
      if (processed_points() == POINTS) { // perform the convergence check only when all points are processed
        var b:Boolean = true;
        for (var j:Long = 0; j < CLUSTERS*DIM; ++j)
          if (Math.abs(old_central_clusters(j) - central_clusters(j)) > 0.0001) { b = false; break; }
        if (b) break; // exit the iteration if converged
      }
    } // end of for (iter)

    // print the result in the central_clusters array
    for (var d:Long = 0; d < DIM; ++d) {
      for (var k:Long = 0; k < CLUSTERS; ++k) Console.OUT.printf("%10.8f ", central_clusters(k*DIM+d));
      Console.OUT.println("<--- dim" + d);
    }
  } // end of main
}
```

Figure 9. Fault-tolerant KMeans.

```
import x10.regionarray.*;
class ResilientHeatTransfer {
  static val N = 20;            // size of grid
  static val ITERATIONS = 1000; // number of maximum iterations
  static val livePlaces = new x10.util.ArrayList[Place](); // set of live places
  static val restore_needed = new Cell[Boolean](false);    // flag to indicate restoration is necessary

  public static def main(args:Rail[String]) {
    for (pl in Place.places()) livePlaces.add(pl);
    val BigR = Region.make(0..(N+1), 0..(N+1)); // 2-dimensional region which includes surroundings
    val SmallR = Region.make(1..N, 1..N);       // 2-dimensional NxN region, which does not include surroundings
    val LastRow = Region.make(0..0, 1..N);      // heated area at the top
    // create data which will be recreated at place death
    var BigD:Dist(2) = Dist.makeBlock(BigR, 0, new SparsePlaceGroup(livePlaces.toRail()));
    var SmallD:Dist(2) = BigD|SmallR;
    var D_Base:Dist = Dist.makeUnique(SmallD.places());
    // create distributed arrays (each element holds a heat value and the LastRow area is always 1.0)
    val A = ResilientDistArray.make[Double](BigD, (p:Point)=>{ LastRow.contains(p) ? 1.0 : 0.0 });
    val Temp = ResilientDistArray.make[Double](BigD); // a DistArray to hold newly calculated values temporarily
    val Scratch = ResilientDistArray.make[Double](BigD);
    A.snapshot(); // create the initial snapshot
    for (iter in 1..ITERATIONS) { Console.OUT.println("Iteration " + iter); // iterate until the result converges
      try {
        // 1. if necessary, restore data from the snapshot
        if (restore_needed()) {
          // recreate Dist over the remaining live places
          BigD = Dist.makeBlock(BigR, 0, new SparsePlaceGroup(livePlaces.toRail()));
          SmallD = BigD|SmallR; D_Base = Dist.makeUnique(SmallD.places());
          A.restore(BigD); // reconstruct DistArray with the new Dist, and restore elements from the snapshot
          Temp.remake(BigD); Scratch.remake(BigD);
          restore_needed() = false;
        }
        // 2. core part of the heat transfer computation
        val D = SmallD;
        finish ateach (z in D_Base) { // distributed processing at each place
          for (p:Point(2) in D|here) { // process the points of this place
            val [x,y] = p;  // stencil computation, average of surrounding four points becomes the new heat value
            Temp(p) = ( (at (A.dist(x-1,y)) A(x-1,y)) + (at (A.dist(x+1,y)) A(x+1,y))
                      + (at (A.dist(x,y-1)) A(x,y-1)) + (at (A.dist(x,y+1)) A(x,y+1)) ) / 4;
        } }
        // 3. check the convergence
        val delta = A.map(Scratch, Temp, D.region, (a:Double,b:Double)=>Math.abs(a-b))
                     .reduce((a:Double,b:Double)=>Math.max(a,b), 0.0);
        Temp.map(A, Temp, D.region, (a:Double,b:Double)=>a); // copy the new results in Temp to A in parallel
        if (delta <= 0.0001) break; // exit the iteration if converged
        // 4. create a snapshot at every 10th iteration
        if (iter % 10 == 0) A.snapshot();
      } catch (e:Exception) { processException(e); } // process an exception
    } // end of for (iter)

    // print the result in the distributed array A
    for ([x] in A.region.projection(0)) {
      for ([y] in A.region.projection(1)) Console.OUT.printf("%5.3f ", at (A.dist(x,y)) A(x,y));
      Console.OUT.println();
    }
  } // end of main

  // process an exception. for DPE, livePlaces is updated and restore_needed flag is set
  private static def processException(e:Exception) {
    if (e instanceof DeadPlaceException) {
      val deadPlace = (e as DeadPlaceException).place;
      livePlaces.remove(deadPlace); restore_needed() = true;
    } else if (e instanceof MultipleExceptions) {
      val exceptions = (e as MultipleExceptions).exceptions();
      for (ec in exceptions) processException(ec);
    } else throw e; // just throw exceptions other than DeadPlaceException
  }
}
```

Figure 10. Fault-tolerant HeatTransfer.

Appendix

The complete code of KMeans and HeatTransfer is shown in Figures 9 and 10. These programs can be compiled and executed by the latest X10 distribution, 2.4.2. Similar programs are also included in the distribution, including MontePi and multiple Resilient DistArray implementations, under the directory `samples/resiliency/`. Refer to the `README.txt` file in the directory for details.