Resilient X10 and Fault Tolerant Application Framework

2015/01/29
Kiyokuni Kawachiya
IBM Research - Tokyo

This work was funded in part by the Air Force Office of Scientific Research under Contract No. FA8750-13-C-0052.
Programming Language X10

- X10 is a programming language that supports parallel/distributed computing internally

- Parallel and distributed Hello World in X10

```x10
class HelloWorld {
    public static def main(args: Rail[String]) {
        finish for (pl in Place.places()) {
            at (pl) async {  // parallel distributed exec in each place
                Console.OUT.println("Hello from "+ here);
            }
        } // end of finish, wait for the execution in all places
    }
}
```

- Compilation and execution example using Native X10 →

  ```bash
  $ x10c++ HelloWorld.x10 -o HelloWorld  # compile
  $ X10_NPLACES=4 runx10 HelloWorld  # execution
  Hello from Place(3)
  Hello from Place(0)
  Hello from Place(2)
  Hello from Place(1)
  ```

Executed at each node (place) in a parallel and distributed manner
Asynchronous Partitioned Global Address Space

- A global address space is divided into multiple *places* (≈ computing nodes)
  - Each place can contain *activities* and *objects*
- An activity (≈ thread) is created by *async*, and can move to another place by *at*
- An object belongs to a specific place, but can be *remotely referenced* from other places
  - To access a remote reference, activities must move to its home place
- *DistArray* is a data structure whose elements are scattered over multiple places
- *PlaceLocalHandle* provides per-place data
If a Computing Node Failed ...

Consider the case Place 1 (’s node) dies

- Activities, objects, and part of DistArrays in the dead place are lost
  - This causes the abort of the entire X10 processing in standard X10
- However in PGAS model, it is relatively easy to localize the impact of place death
  - Objects in other places are still alive, although remote references become inaccessible
  - Can continue the execution using the remaining nodes (places)  \( \Rightarrow \) Resilient X10
Resilient X10 – Extension for Fault Tolerance

- Stores activities’ critical information in a “Resilient Storage”
  - The most stable implementation uses Place 0 for this purpose
- Throws a new exception, DeadPlaceException (DPE), for a place death
  - If an activity is being moved to the place, corresponding `at` throws the DPE
  - If an activity is asynchronously executed in the dead place by `async`, governing `finish` will throw a MultipleExceptions which contains the DPE

- A simple fault-tolerant program which just reports node failures

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

If the target place (pl) dies, `at` statement throws a DPE, which is caught here
Utilization of the Rooted Exception Model

- In X10, exceptions thrown from asynchronous activities can be caught

```scala
class HelloWorld {
    public static def main(args:Rail[String]) {
        try {
            finish for (pl in Place.places()) {
                at (pl) async {
                    // parallel distributed exec in each place
                    Console.OUT.println("Hello from " + here);
                    do_something();
                }
            } // end of finish, wait for the execution in all places
        } catch (es:MultipleExceptions) { for (e in es.exceptions()) ... }
    }
}
```

- The `finish` governing the activity (async) receives the exception(s), and throws a `MultipleExceptions`... Rooted Exception Model
  - By enclosing a `finish` with `try ~ catch`, async exceptions can be caught
    - `DeadPlaceException` can be caught with the same mechanism
Resilient Applications and Data Structures

- The DeadPlaceException notification (and some support methods) are sufficient to add fault tolerance to existing distributed X10.

- However, it is necessary to understand the structure of each application:
  - How the application is doing the distributed processing?
  - How the execution can be continued after a node failure?

→ We prepared various resilient applications and libraries

- Three basic methods to add resiliency
  1. Resilient MontePi – *ignore* failures and use the results from the remaining nodes
  2. Resilient KMeans – *reassign* the failed node’s work to the remaining nodes
  3. Resilient HeatTransfer – *restore* the computation from a periodic snapshot

- Data structures for resiliency
  4. Resilient DistArray
  5. Resilient PlaceLocalHandle
  6. ResilientStore – resilient key/value store to save data resiliently

- Resilient MapReduce
  7. Resilient M3R Lite MapReduce Engine
  8. Resilient KMeans job for Resilient M3R Lite

---

M3R: Main Memory MapReduce
- A MapReduce engine written in X10
- M3R Lite is a simple version of M3R written by Vijay Saraswat
(1) MontePi – Computing π with the Monte Carlo Method

Overview

- Try ITERS times at each place, and update the result at Place 0
- Place death is simply ignored
  - The result may become less accurate, but it is still correct

```scala
class ResilientMontePi {
  public static def main (args:Rail[String]) {
    finish for (p in Place.places()) async {
      try {
        at (p) {
          val md = new x10.util.Random(System.nanoTime());
          var c:Long = 0;
          for (iter in 1..ITERS) { // ITERS trials per place
            val x = md.nextDouble(), y = md.nextDouble();
            if (x*x + y*y <= 1.0) c++; // if inside the circle
          }
          val count = c;
          at (result) atomic { // update the global result
            val r = result(); r() = Pair(r().first+count, r().second+ITERS);
          }
        }
      } catch (e:DeadPlaceException) {/* just ignore place death */}
    } // end of finish, wait for the execution in all places
  }   }

// calculate the value of π and print it */
```
(2) KMeans – Clustering Points by K-Means

**Overview**
- Each place processes assigned points, and iterates until convergence
  - Don’t assign the work to dead place(s)
    - The work is reassigned to remaining places
- Place death is ignored
  - Partial results are still utilized

```scala
class ResilientKMeans {
    public static def main(args:Rail[String]) {
        for (iter in 1..ITERATIONS) {
            /* deliver current cluster values to other places */
            val numAvail = Place.MAX_PLACES - Place.numDead();
            val div = POINTS / numAvail; // share for each place
            val rem = POINTS % numAvail; // 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;
                    at (pl) async { /* process [start,end), and return the data */}
                    start = end;
                } // end of finish, wait for the execution in all places
            } catch (es:MultipleExceptions) { /* just ignore place death */
                /* compute new cluster values, and exit if converged */
            } // end of for (iter)
            /* print the result */
        }
    }
}
```

Assign the work only to live nodes
(3) HeatTransfer – Computing Heat Diffusion

```scala
import x10.resilient.regionarray.DistArray;
class ResilientHeatTransfer {
    static val livePlaces = new ArrayList[Place]();
    static val restore_needed = new Cell[Boolean](false);
    public static def main(args:Rail[String]) {
        val A = DistArray.make[Double](BigD, ...); // create a DistArray
        A.snapshot(); // create the initial snapshot
        for (iter in 1..ITERATIONS) { // iterate until convergence
            try {
                if (restore_needed()) {
                    // if some places died
                    val livePG = new SparsePlaceGroup(livePlaces.toRail());
                    BigD = Dist.makeBlock(BigR, 0, livePG); // recreate Dist, and
                    A.restore(BigD); // restore elements from the snapshot
                    restore_needed() = false;
                }
            } finally {
                foreach (z in D_Base) {
                    /* compute new heat values for A's local elements */
                }
                /* if converged, exit the for loop */
                if (iter % 10 == 0) A.snapshot(); // create a snapshot at every 10th iter.
            }
        }
        catch (e:Exception) { processException(e); }
    }
}
```

Overview

- A 2D DistArray holds the heat values of grid points
- Each place computes heat diffusion for its local elements
- Create a snapshot of the DistArray at every 10th iteration
- Upon place death, the DistArray is restored from the snapshot

Remove the dead place from the livePlaces list and set the restore_needed flag
(4) Resilient DistArray

- An extended DistArray which supports *snapshot* and *reconfiguration*

```java
package x10.resilient.regionarray;
public class DistArray[T] ... {
    // Normal DistArray interfaces, plus the followings:

    public static def make[T](dist:Dist, init:(Point)=>T) : DistArray[T];
    public static def make[T](dist:Dist){T haszero} : DistArray[T];
    public final operator this(pt:Point) : T; // read element
    public final operator this(pt:Point)=(v:T) : T; // set element
    public final def map[S,U](dst:DistArray[S], src:DistArray[U],
        filter:Region, op:(T,U)=>S) : DistArray[S];
    public final def reduce(op:(T,T)=>T, unit:T) : T;
    // Create a snapshot
    public def snapshot() { snapshot_try(); snapshot_commit(); }
    public def snapshot_try() : void;
    public def snapshot_commit() : void;
    // Reconstruct the DistArray with new Dist
    public def restore(newDist:Dist) : void;
    public def remake(newDist:Dist, init:(Point)=>T) : void;
    public def remake(newDist:Dist){T haszero} : void;
}
```

Interface overview
- Normal DistArray interfaces, plus the followings:
  - *snapshot()*
  - Dump the element values into the Resilient Storage
  - *restore(newDist)*
  - Reconstruct the DistArray over live places, and restore the snapshot
(5) Resilient PlaceLocalHandle

- An extended PlaceLocalHandle which supports *snapshot* and *restoration*

```java
class PlaceLocalHandle<T> ... {
    public static def make(pg:PlaceGroup, init:()=>T): PlaceLocalHandle[T];
    public operator this():T;

    // Create a snapshot
    public def snapshot(): { snapshot_try(); snapshot_commit(); };
    public def snapshot_try():void;
    public def snapshot_commit():void;

    // Reconstruct the PLH over the new PlaceGroup
    public def restore(newPg:PlaceGroup):void;
    public def remake(newPg:PlaceGroup, init:()=>T): PlaceLocalHandle[T];
}
```

Example code is found at samples/resiliency/ResilientPlhHeatTransfer.x10
(6) ResilientStore

- A resilient key/value store to save data resiliently
  - Data is stored resiliently, and can be retrieved from any place, even if some places are dead
  - Used to implement resilient DistArray and PLH to keep snapshots

```java
package x10.resilient.util;
public abstract class ResilientStoreForApp[K,V] ... {
    public static def make[K,V](): ResilientStoreForApp[K,V];
    ...
    // HashMap-like I/F
    public def put(key:K, value:V):void;
    public def getOrThrow(key:K):V;
    public def get(key:K):V;
    public def getOrElse(key:K, orelse:V):V;
}
```

- Several implementations are provided
  - Place0-based, Hazelcast-based, ...
  - Can be selected by “X10_RESILIENT_STORE_MODE”
(7) Resilient MapReduce Engine

MapReduce Engine “M3R Lite” now supports place deaths

- When a place is dead during an *iterative* map-reduce execution, next iteration will be done excluding the dead place.
  - If an environment variable `M3RLITE_NSPARES=<n>` is specified, n places are reserved as spare places, which will replace the dead place.
  - If there is no spare places remaining, the dead place is just excluded and the iteration can continue on the reduced number of places. This can be disabled by an environment variable `M3RLITE_NOSHRINK=1`. In this case, the execution will fail when there is no remaining spare place.

- Some methods are added to the Engine
  - `numLivePlaces()` returns the number of places used for this iteration
  - `placeIndex(Place)` returns the index of the specified place
  - `iterationNumber()` returns current iteration number.
  - `iterationFailed()` returns whether the last iteration failed. Should be called from `stop()`.

https://svn.code.sf.net/p/x10/code/applications/trunk/m3rlite/src/com/ibm/m3rlite/ResilientEngine.x10
Writing a Job for Resilient M3R Lite

The Job interface is basically same as that of (non-resilient) M3R Lite, but the Job is responsible to:

- Keep the source data resiliently (e.g. by using ResilientStore)
- Divide the source data appropriately using placeIndex and numLivePlaces
- Not use Place.places() or numPlaces() during the execution
- Return false in stop() if all data are not processed in that iteration
- Jobs implemented: KMeans, HMMTrainer, SparseMatVecMult

```java
public interface Job[K1,V1,K2,V2,K3,V3] {
    // Invoked in each place to obtain the data
    def source():Iterable<Pair[K1,V1]];
    // Translate a (K1, V1) pair into zero or more (K2, V2) pairs
    def mapper(K1, V1, (K2,V2)=>void):void;
    // Return the partition to which the key belongs
    def partition(k:K2):Long;
    // Take a K2 and a sequence of V2 and add resulting (K3, V3) pairs
    def reducer(K2,Iterable[V2], ArrayList<Pair[K3,V3]]):void;
    // Consume the (K3, V3) pairs supplied by the reduce
    def sink(Iterable<Pair[K3,V3]]):void;
    // Return true if the job should terminate
    def stop():Boolean;
}
```
(8) KMeans on Resilient M3R Lite

source
mapper
partition
reducer
sink
stop

Get cluster coords
Calculate my part
(e.g. 0~6665)
and cache the data

For each point,
determine the nearest cluster

Caculate new cluster coords
Put the new cluster coords to ResilientStore

if (!iterationFailed)
update clusters from ResStore
if (converged) exit

Get cluster coords
Calculate my part
(e.g. 6666~13332)
and cache the data

For each point,
determine the nearest cluster

Caculate new cluster coords
Put the new cluster coords to ResilientStore

Get cluster coords
Calculate my part
(e.g. 13333~19999)
and cache the data

For each point,
determine the nearest cluster

Caculate new cluster coords
Put the new cluster coords to ResilientStore

ResilientStore

Points data

Points 0~9999

Points 10000~19999

Tentative clustering results

New cluster 0

New cluster 1

New cluster 2

New cluster 3

(Point ID, Coords)

(Cluster ID, Point’s Coords)

(Cluster ID, New Coords)
Performance Evaluation

- Measured with KMeans on Resilient M3R Lite + Resilient Store
  - 1,000,000 of 3D points were categorized into 8 clusters
  - 7 places (+ 1 spare) were used for the computation
  - Hazelcast was used for resilient finish/resilient store

Observation

- Resilient X10 (HC-based) was ~9% slower than normal X10, in this case
- However, it could survive a place failure and continue the calculation
Summary

Introduced how to write resilient applications on Resilient X10

- Three basic methods to add resiliency
  1. Resilient MontePi – just ignore failures
  2. Resilient KMeans – reassign the failed node’s work
  3. Resilient HeatTransfer – restore the computation from snapshot

- Data structures for resiliency
  4. Resilient DistArray
  5. Resilient PlaceLocalHandle
  6. ResilientStore – resilient key/value store

- Resilient MapReduce
  7. Resilient M3R Lite MapReduce Engine
  8. Resilient KMeans job for Resilient M3R Lite
Additional Information about Resilient X10

- The Resilient X10 function is included in X10 2.4.1 and later versions
  - Can be enabled by specifying “X10_RESILIENT_MODE=1”
  - Can run with either of Native X10 and Managed X10
    - The communication layer is limited to sockets
  - Sample codes exist under “samples/resiliency/”
    - Refer to README.txt in the directory for details
  - Resilient M3R Lite code is under https://svn.code.sf.net/p/x10/code/applications/trunk/m3rlite/

- Related papers
  **Resilient X10: Efficient Failure-Aware Programming**

  **Semantics of (Resilient) X10**

  **Writing Fault-Tolerant Applications Using Resilient X10**