Writing Fault-Tolerant Applications Using Resilient X10

2014/06/12
Kiyokuni Kawachiya (presentation by Mikio Takeuchi)
IBM Research - Tokyo

This work was funded in part by the Air Force Office of Scientific Research under Contract No. FA8750-13-C-0052.
Writing Fault-Tolerant Applications Using Resilient X10

Contents

▪ X10 overview
  – Extension for fault tolerance ... Resilient X10
▪ X10 applications with fault tolerance
  – Computing $\pi$ with the Monte Carlo method ... MontePi
  – Clustering points by K-Means ... Kmeans
  – Computing heat diffusion ... HeatTransfer
▪ Evaluation
  – Modification size, performance, and fault tolerance
▪ Summary
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
  }
}
```

Fig 3

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

```java
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
Writing Fault-Tolerant Applications

- 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?

→ Introduce three methods to add fault tolerance
  (a) Ignore failures and use the results from the remaining nodes
  (b) Reassign the failed node’s work to the remaining nodes
  (c) Restore the computation from a periodic snapshot. (b)+checkpointing
(a) MontePi – Computing $\pi$ with the Monte Carlo Method

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 $\pi$ and print it */
    }

In case of node failure

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 valid

Fig 4
(b) 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

Assign the work only to live nodes

```scala
class ResilientKMeans {
  public static def main(args: Rail[String]) {
    for (iter in 1..ITERATIONS) {
      // iterate until convergence
      /* 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 */
      }
    }
  }
}
```
Overview

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

Remove the dead place from the livePlaces list and set the restore_needed flag

```scala
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 = ResilientDistArray.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;
        }
        finish at each (z in D_Base) { // distributed processing
          /* compute new heat values for A's local elements */
          Temp = ((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;
        }
        /* if converged, exit the for loop */
        if (iter % 10 == 0) A.snapshot(); // create a snapshot at every 10th iter.
      } catch (e:Exception) { processException(e); } 
    } // end of for (iter)
    /* print the result */
  }
}
```
Resilient DistArray – Fault Tolerant DistArray

- An extended DistArray which supports snapshot and reconfiguration

```java
public class ResilientDistArray<T> ... {
    public static def make[T](dist:Dist, init:(Point)=>T) : ResilientDistArray[T];
    public static def make[T](dist:Dist{T haszero}) : ResilientDistArray[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:ResilientDistArray[S], src:ResilientDistArray[U],
        filter:Region, op:(T,U)=>S) : ResilientDistArray[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

Fig 6
Evaluation – Modification Amount

- Modifications necessary to add fault tolerance to the base codes

<table>
<thead>
<tr>
<th>Program</th>
<th>Code size</th>
<th>Mod. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MontePi</td>
<td>30 lines</td>
<td>4 lines</td>
</tr>
<tr>
<td>KMeans</td>
<td>93 lines</td>
<td>12 lines</td>
</tr>
<tr>
<td>HeatTransfer</td>
<td>68 lines</td>
<td>24 lines</td>
</tr>
<tr>
<td>(ResilientDistArray)</td>
<td>~200 lines</td>
<td>—</td>
</tr>
</tbody>
</table>

Consideration

- Fault tolerance could be added with very small modifications
  - The modified code can still run on standard X10, as long as node failure does not occur
- Modification ratio of HeatTransfer is relatively large, but
  - Larger DistArray applications can be made fault tolerant using the same approach, and the modification ratio will be smaller
  - Half of the modifications are in the exception-handling code of processException, which can be provided as a support library
Evaluation – Performance

- Relative execution times of base code and fault-tolerant code, on Standard and Resilient X10

Consideration
- On Standard X10
  - For MontePi and KMeans, almost no overheads were observed
  - For HeatTransfer, FT version took 9% more time, because of the cost of the periodic snapshots
- On Resilient X10
  - For MontePi and KMeans, there were 2.2~9.0% overheads
  - For HeatTransfer, 6x slowdown mainly because at is invoked too frequently
Evaluation – Fault Tolerance

- Behavior when places are killed

<table>
<thead>
<tr>
<th></th>
<th>Base code</th>
<th>Fault-tolerant code</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Standard X10</td>
<td>Aborted</td>
<td>Aborted</td>
</tr>
<tr>
<td>On Resilient X10</td>
<td>End with DPE</td>
<td>Survived</td>
</tr>
</tbody>
</table>

→ Fault tolerance was achieved by the combination of Resilient X10 and fault-tolerant applications

- Effects caused by place deaths
  - When 4 places among the 8 places were killed
    - Deviation of MontePi result increased from 0.0008% to 0.002%
    - But the execution time did not increase
  - 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
Conclusions

Summary

✓ Introduced three methods of adding fault tolerance to existing applications
  (a) Ignore failures and use the results from the remaining nodes ... MontePi
  (b) Reassign the failed node’s work to the remaining nodes ... KMeans
  (c) Restore the computation from a periodic snapshot ... HeatTransfer

✓ Evaluated the fault-tolerant applications in a real distributed environment
  – Very small modifications to add fault tolerance
  – 2.2~9.0% execution overhead (but 6x slowdown in case of too frequent at)
  – 11~14% additional overhead when 1 among 8 places was lost

Future work

  – Reduce the overhead – both in Resilient X10 and fault-tolerant applications
  – Make fault-tolerant versions of larger X10 applications [12,19]
Additional Information about Resilient X10

- The Resilient X10 function is included as a technology preview in X10 2.4.3 (released in May 2014)
  - 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

- Related papers

  [2] Semantics of (Resilient) X10
  Crafa, S., Cunningham, D., Saraswat, V., Shinnar, A. and Tardieu, O.
BACKUP
Execution Example – HeatTransfer (10x10)

$ cd X10/242/samples/resiliency
$ x10c++ -O NO_CHECKS
ResilientHeatTransfer.x10
-o ResilientHeatTransfer
$ X10_NPLACES=8
X10_RESILIENT_MODE=1
runx10 ResilientHeatTransfer 10
HeatTransfer for 10x10, epsilon=1.0E-5

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

X10_RESILIENT_STORE_MODE=0
X10_RESILIENT_STORE_VERBOSE=0
— Iteration: 1
delta=0.25
— Iteration: 2
delta=0.125
:  
— Iteration: 10
delta=0.023306892944336
Create a snapshot at iteration 10
— Iteration: 11
delta=0.020451545715332
:  
— Iteration: 38
delta=0.003633990233121
— Iteration: 39  Place 2 was killed
Place 2 exited unexpectedly with signal: Terminated
MultipleExceptions size=2
DeadPlaceException thrown from Place(2)
DeadPlaceException thrown from Place(2)
— Iteration: 40
Create new Dist over available 7 places
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7 7

Restore from a snapshot at iteration 30
delta=0.005177850168275
Create a snapshot at iteration 40
— Iteration: 41
delta=0.004930303185298
:  
— Iteration: 85  Place 7 was killed
Place 7 exited unexpectedly with signal: Terminated
MultipleExceptions size=1
DeadPlaceException thrown from Place(7)
— Iteration: 86
Create new Dist over available 6 places
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7 7

Restore from a snapshot at iteration 80
delta=0.003633990233121
— Iteration: 87
delta=0.003633990233121
:  
— Iteration: 194
delta=0.003633990233121
Result converged

— Result
0.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 0.000
0.000 0.491 0.679 0.761 0.799 0.815 0.815 0.799 0.761 0.679 0.491 0.000
0.000 0.285 0.463 0.566 0.622 0.646 0.646 0.622 0.566 0.463 0.285 0.000
0.000 0.184 0.324 0.419 0.475 0.501 0.501 0.475 0.419 0.324 0.184 0.000
0.000 0.126 0.232 0.309 0.358 0.382 0.382 0.358 0.309 0.232 0.126 0.000
0.000 0.089 0.167 0.227 0.267 0.287 0.287 0.267 0.227 0.167 0.089 0.000
0.000 0.064 0.121 0.166 0.197 0.212 0.212 0.197 0.166 0.121 0.064 0.000
0.000 0.045 0.086 0.118 0.141 0.153 0.153 0.141 0.118 0.086 0.045 0.000
0.000 0.031 0.058 0.081 0.097 0.105 0.105 0.097 0.081 0.058 0.031 0.000
0.000 0.019 0.036 0.051 0.061 0.066 0.066 0.061 0.051 0.036 0.019 0.000
0.000 0.009 0.017 0.024 0.029 0.032 0.032 0.029 0.024 0.017 0.009 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
import x10.regionarray.*;

class ResilientKMeans {
    static val DIM = 4;
    // number of dimensions
    static val POINTS = 16000000; // 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_cluster_counts];
        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;
                        }
                    }
                } 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
        var 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() += tmp_new_clusters(j);
            for (var j:Long = 0; j < CLUSTERS; ++j) central_cluster_counts_gr() += tmp_cluster_counts(j);
            processed_points_gr() += tmp_processed_points;
        }
    } // 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
ResilientHeatTransfer

```scala
import x10.regionarray.*;

class ResilientHeatTransfer {

  static val N = 20;       // size of grid
  static val ITERATIONS = 10000;   // 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 N x N 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[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) { // 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
if (iter % 10 == 0) A.snapshot();
} catch (e:Exception) { processException(e); } // process an exception

// 4. create a snapshot at every 10th iteration

// 5. 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
}
Programming Language X10

- X10 [25] is a programming language that supports parallel/distributed computing internally

- Parallel and distributed Hello World in X10

```java
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)
```

Fig 2

Executed at each node (place) in a parallel and distributed manner
Execution Model of X10 – Asynchronous PGAS

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
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) → Resilient X10 [2,3]
DistArray – A Distributed Array Library

- DistArray – An array whose elements are scattered over multiple places
  - Suitable for SPMD-style processing

**Example** – Compute $1^2 + 2^2 + \cdots + 1000^2$ using multiple places

```scala
public class DistArrayExample {
    public static def main(Rail[String]) {
        val R = Region.make(1..1000);
        val D = Dist.makeBlock(R, 0, PlaceGroup.WORLD);
        val A = DistArray.make[Long](D, ([i]:Point)=>i);
        val tmp = new Array[Long](Place.MAX_PLACES);
        finish for (pl in Place.places()) async {
            tmp(pl.id) = at (pl) {
                var s:Long = 0;
                for (p:Point in D|here) s += A(p)*A(p); s;
            } // end of finish, wait for the execution in all places
            result = tmp.reduce((a:Long,b:Long)=>a+b, 0);
            Console.OUT.println(result); // -> 333833500
        }
    }
}
```

- Upon a node failure, the DistArray elements in the dead place are lost

**Overview**

- Create a DistArray D with initial values 1~1000
- Each place processes its local elements (x), and calculates the sum of $x^2$
- Sum up the all results

Local part of the DistArray is processed here