Resilient Distributed Programming Language X10

2015/06/26
Kiyokuni (KIYO) Kawachiya
IBM Research - Tokyo
About the Speaker

Name: Kiyokuni Kawachiya (http://ibm.co/kawatiya)

- Research Staff Member of IBM Research - Tokyo
  - Also a Manager of Deep Computing & Analytics Group

- Joined IBM in 1987 after receiving M.S. degree from U. Tokyo
  - Received Ph.D. degree from Keio University in 2005

- Main interest is “System Software”
  - Operating systems, virtualization, programming languages, ....
  - Has been working on multiprocessor OS, multimedia systems, mobile information devices, Java execution runtime, ...
  - Recently working for High-Performance Computing and its programming environment such as X10 ← Today’s Topic
Today’s Talk

Title: Resilient Distributed Programming Language X10

- X10 – a programming language being developed by IBM Research
  - Open source project (see http://x10-lang.org/)
  - Supports “distributed processing” using multiple computing nodes

- This talk introduces X10, focusing on its distributed processing functions
  - X10 at a glance
    - “APGAS” execution model – place and activity
    - Distributed “Hello World”
  - Distributed processing with X10
    - Primitives – at, async, finish
    - Data structures for distributed processing
  - Some advanced topics
    - Distributed garbage collection
    - Fault tolerance support by “Resilient X10”
X10 at a Glance
The Manycore Programming Challenge

- Programmers cannot cope with thousands of threads and complex data flows using existing programming models

→ How to expose the parallel distributed environment to programmers?

Yesterday

- Single Core
- Single Thread
- 100% Serial Programming

Today

- Multicore (2-16)
- Multithread (10s)
- 80/20 Serial/Parallel Programming

Tomorrow

- Manycore (32-100s)
- 20/80 Serial/Parallel Programming

Threading model breaks as complexity exceeds programmer capability

From “Global Technology Outlook 2010” by IBM Research
How to Expose the Parallel Distributed Environment?

Partitioned Global Address Space (PGAS)
- A global address space is divided into multiple places (≈ computer)
  - Place is an abstraction of locality, and can be heterogeneous (PPE, SPE, GPU, ...)
- An object belongs to a place where it is created
  - Distributed data structures (e.g. co-array) are also provided
- Object cannot be accessed from other places
  - But can be remotely referenced from other places

\[ \text{X10} \approx \text{PGAS implementation of Java-like language} \]
- Also supports asynchronous activity execution → “Asynchronous PGAS (APGAS)”
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 by **GlobalRef**
  - 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
“Hello World” in X10

- **X10 Program**

```java
class MyHello {
    public static def main(Rail[String]) {
        finish for (pl in Place.places()) {
            at (pl) async Console.OUT.println("Hello World from " + here);
        }
    }
}
```

- **Compile and Execute**

```
$ x10c MyHello.x10 # compile
ls
MyHello.x10 MyHello$$Main.class
MyHello.java MyHello$$Closure$0.class
MyHello.class

$ X10_NPLACES=4 x10 MyHello # execute
Hello World from Place(3)
Hello World from Place(0)
Hello World from Place(1)
Hello World from Place(2)
```

- X10 program is translated into Java (or C+) source code and further compiled.
- Executed asynchronously on each place.
- For C++ back-end:
  ```
  $ x10c++ -o MyHello MyHello.x10
  $ X10_NPLACES=4 runx10 MyHello
  ```

Parallel distributed execution by creating an activity at each place.
X10DT – An IDE for X10 App Development

Integrated Development Environment (IDE) for X10 on Eclipse
X10 Development

X10 is an Open Source Project
– Project page: http://x10-lang.org/
– Originally started as part of HPCS project funded by DARPA
– Eclipse Public License

▪ Source Code Management
  – SourceForge (svn) https://svn.code.sf.net/p/x10/code
  – JIRA (bug tracking) https://xtenlang.atlassian.net/projects/XTENLANG

▪ X10 Community
  – Mailing Lists http://sourceforge.net/p/x10/mailman/
    • For users x10-users@lists.sourceforge.net
    • For developers x10-core@lists.sourceforge.net
  – X10 Workshop colocated with PLDI
    • X10’15 just held on June 14 at Portland, Oregon
X10 Project Page: [http://x10-lang.org/](http://x10-lang.org/)
マルチコア時代のプログラミング言語「X10」

マルチコア時代のプログラミング

ロボット工学の性能向上と長く、クロックの高速化によるプログラムの改善によって進化してきた。しかし、シングルプロセッサの性能向上は限界もあり、オペレーティングシステムの開発に至り、リアルタイム性を向上させる新技術の導入が求められている。このため、新しいプログラミング言語が必要である。X10は、IBM Researchが開発した新しいプログラミング言語で、ローカルコンピューティングの環境に適したソリューションを提供するものである。この言語は、並列処理やマルチコア環境での処理を効率的に実装するための機能を備えている。

X10の概要

X10は、IBM Researchが開発している新しいプログラミング言語で、Javaベースのプログラミング言語で、Javaの特徴を最大限に活かした機能を持つ言語である。X10は、インタラクティブな実行環境を提供し、開発者がより効率的にプログラムを書くことができる。また、X10はライブラリ機能を強化し、開発者がより簡単にプログラムを書くことができる。

情報処理, Vol. 52, No. 3, pp. 342-356 (2011/03)
Programming with X10
Overview of X10 Programming

X10 is a modern type-safe Object-Oriented language
- Supports scalable programming for a heterogeneous parallel distributed environment, where various processors (CPUs, GPUs, ...) are interconnected

- Java-like language
  - Object oriented, statically typed
  - Single inheritance (+ interfaces)
  - Execution starts from “main”
  - Java-like syntax
    - for, while, if, switch, ...

- Differences from Java
  - Scala-like declaration
    - val, var, def, [T], ...
  - Stronger generics support
  - Unsigned numbers support
  - New data types: struct and func
  - Strong type inference, etc.
  - And, parallel distributed processing
  - ...

```java
import x10.util.Pair; // struct to represent a pair
public class MySample[T] {
    val data:T;
    def this(d:T) { data = d; } // constructor
    def get() = data;

    public static def main(args:Rail[String]):void {
        /* Add from 1 to arg */
        val end = (args.size > 0) ? Long.parse(args(0)) : 10;
        var sum:Long = 0;
        for (var i in 1..end) sum += i;
        Console.OUT.printf("Sum of 1-%d: %d¥n", end, sum);

        /* Object creation with generics */
        val obj = new MySample[Double](1.2);
        Console.OUT.println(obj.get());  // -> 1.2

        /* Various data types */
        val pair = Pair[UInt,UInt](3u,4u); // struct
        val func = (i:UInt,j:UInt)=>i*j;   // function
        Console.OUT.println(func(pair.first,pair.second));  // -> 12
    }
}
```

Result:
- Sum of 1–10: 55
- 1.2
- 12
Parallel Distributed Prog.

Parallel processing
- **async S** executes S in a new activity
  - The parent activity does not wait and runs in parallel
  - S can access outside local variables
- **finish S** waits for the termination of all activities created inside S
  - Grandchildren are also waited
  - Exceptions inside S are caught and thrown as a MultipleExceptions

Distributed processing
- **at (pl) S** executes S at Place pl
  - Next line is executed when S finishes
  - S can access outside val variables
- Objects to be accessed in S are deep-copied to the target place
  - Global reference to an object can be created using GlobalRef[T]
  - Activities must move to the home place to access the global object

```scala
class MyFib {
  var r: Int; // in-out parameter
  def this(i: Long) { r = i; }
  def run() {
    if (r < 2) return r; // MyFib(0)==0, MyFib(1)==1
    val f1 = new MyFib(r-1), f2 = new MyFib(r-2);
    finish {
      async f1.run(); // compute MyFib(r-1) in parallel
      f2.run(); // compute MyFib(r-2) by myself
    }
    return (r = f1.r + f2.r); // MyFib(r-1) + MyFib(r-2)
  }
}
```

```scala
public class MyFib {
  var r: Int; // in-out parameter
  def this(i: Long) { r = i; }
  def run() {
    if (r < 2) return r; // MyFib(0)==0, MyFib(1)==1
    val f1 = new MyFib(r-1), f2 = new MyFib(r-2);
    finish {
      async f1.run(); // compute MyFib(r-1) in parallel
      f2.run(); // compute MyFib(r-2) by myself
    }
    return (r = f1.r + f2.r); // MyFib(r-1) + MyFib(r-2)
  }
}
```

```scala
public static def main(args: Array[String]) {
  /* Compute a Fibonacci number in parallel */
  val n = (args.size > 0)? Long.parseLong(args(0)) : 10;
  val f = new MyFib(n); f.run();
  Console.OUT.println("Fib(\n");
  /* Place shift */
  Console.OUT.println(Place.MAX_PLACES); // -> 4 (etc.)
  val r = at (here.next()) new MyFib(10).run();
  Console.OUT.println(r);
  /* Object and GlobalRef */
  val o = new MyFib(0);
  at (here.next()) o.r = 1; // copy of o is modified
  Console.OUT.println(o.r); // -> 0
  val g = GlobalRef[MyFib](o);
  at (here.next()) { at (g.home) g().r = 2; }
  Console.OUT.println(o.r); // -> 2
}
```
X10 can Catch Asynchronous Exceptions

- 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
Example: Computing π with the Monte Carlo Method

```
class MontePi {
    public static def main (args:Rail[String]) {
        finish for (p in Place.places()) {
            async 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);
                }
            }
        }
        // end of finish, wait for the execution in all places
    }
}
```

Overview

- Try ITERS times at each place
- Update the result at Place 0
- When all computations finish, calculate π from the accumulated result

```
val pair = result();
val pi = 4.0 * pair.first / pair.second;
Console.OUT.println("pi="+pi + " (try="+pair.second+")");
```
Data Structures for Distributed Processing
DistArray – Distributed Array

- **DistArray[T]** is an array whose elements are scattering over multiple places
  - Created by DistArray.make(…)
  - The Dist argument provides a mapping from Point to Place

- To access an element, the activity must move to the place the element exists
  - e.g. `at (da.dist(pt)) da(pt)`

```
class DistArrayTest {
  public static def main(Rail[String]) {
    val R = (1..4)*(5..6);
    val D = Dist.makeBlock(R);
    val da = DistArray.make[Int](D, ([i, j]:Point) => i * j );
    for (pt in da)
      Console.OUT.println(at (da.dist(pt)) da(pt));
  }
}
```
DistArray Example

Compute $1^2 + 2^2 + \ldots + 1000^2$

- Initialize DistArray `da` with 1~1000
- Prepare a local array `tmp` to receive a temporary result from each place
- Compute at each place
  - For each element in the place, square it and sum up
- When the computations finish, sum up the results in `tmp`

Note. DistArray provides utility methods to do similar processing
- `map(func:(T)=>T):DistArray[T]`
- `reduce(func:(T,T)=>T, unit:T)`

```java
public class DistArrayExample {
    public static def main(Rail[String]) {
        val D = Dist.makeBlock(1..1000);
        val da = DistArray.make[Int](D, ([i]:Point)=>i);

        val places = da.dist.places();  // Sequence[Place]
        val tmp = new Rail[Int](places.size);
        finish for ([ i ] in 0..(places.size-1)) async {
            tmp(i) = at (places(i)) {
                val a = da | here;  // restriction
                var s:Int = 0;
                for (pt in a) s += a(pt)*a(pt);
                s // return value of at
            };
        }  // end of finish
        var s0:Int = 0; for (pt in tmp) s0 += tmp(pt);
        Console.OUT.println(s0);  // -> 333833500

        /* DistArray.map and reduce */
        val s1 = da0.map((i:Int)=>i*i).reduce((i:Int,j:Int)=>i+j, 0);
        Console.OUT.println(s1);  // -> 333833500
    }
}
```
PlaceLocalHandle – Per-Place Data Holder

- **PlaceLocalHandle[T]** is a handle to hold per-place data
  - Created by `PlaceLocalHandle.make(...)`

```
class PLHTest {
    public static def main(Rail[String]) {
        val plh = PlaceLocalHandle.make[String](Place.places(), () => here.id.toString());
        for (p in Place.places()) {
            val value = at (p) plh();
            Console.OUT.println(val);
        }
    }
}
```

- “plh()” returns different data at each place

- DistArray uses PlaceLocalHandle to hold data for each place
GlobalRef – Global Reference to an Object

- GlobalRef[T] creates a global reference to an object
  - Created by GlobalRef(obj)
  - The object referenced by a GlobalRef can be accessed by moving to the gref’s home place

- PlaceLocalHandle uses GlobalRef to detect that the plh is no longer used
  - If this GlobalRef is collected, data held at each place is deleted

→ GlobalRef is the core mechanism for other distributed data structures

```scala
class GRefExample {
  static class ResultBox { var value:Long = 0; }

  public static def main(Rail[String]) {
    val place1 = here.next();
    val o = new ResultBox();
    val g = GlobalRef[ResultBox](o);
    finish {
      at(place1).async { // create an activity at place1
        val r = do_long_calculation(g);
        at(g).g().value = r; // set the result through GlobalRef
      }
      do_some_calculation_locally();
    } // end of finish
    Console.OUT.println(o.value);
  }
}
```
Summary – Distributed Processing with X10

- Primitives for distributed processing
  - Place is an abstraction of a computing node
  - `at` moves current activity to another place
  - `async` creates a new activity
  - `finish` waits for activities

- Data structures for distributed processing
  - DistArray (distributed array)
  - PlaceLocalHandle (per-place data holder)
  - GlobalRef (global reference to an object)
Distributed GC
**Distributed GC is Supported in Managed X10**

**Managed X10** = X10 implementation on Java (i.e. managed runtime)
- X10 program is translated into Java source code and compiled into Java bytecode, then executed on multiple Java VMs.

![Compilation Flow of X10](image-url)
Remote Reference using GlobalRef

GlobalRef is a special struct to hold a global reference to an object
- Created by “GlobalRef[T](obj)” and cannot be modified
- The object can be accessed by “at (g) g()…”

Figure. Distributed Processing with GlobalRef.
Garbage Collection in Managed X10

- X10 data is represented by Java objects and collected by each JVM’s GC
  - However, remote reference is *not* a reference in the JVM level

- In old X10, remotely-referenced (globalized) objects were registered into a management table and *never collected*

  ➔ We needed better implementation
Data Structures for Distributed GC

- **GOT** tracks a globalized object
  - weakRef is used to detect the object’s collection
  - strongRef is used to prohibit the object’s collection
    - strongRef is cleared if remoteCount is zero, which enables the object’s collection
  - remoteCount holds the total value of weightCounts of the corresponding RRTs

- **RRT** tracks a remote reference
  - weakRef is used to detect the GlobalRef’s collection
    - A notification is sent to home place to decrease the remoteCount of GOT
  - weightCount is used to reduce communication with the home place
Behavior of the Distributed GC

Creating and using a GlobalRef

```scala
class GRefExample {
  static class ResultBox { var value: Long = 0; }
  public static def main(Rail[String]) {
    val place1 = here.next();
    val o = new ResultBox();
    val g = GlobalRef[ResultBox](o); // g
    finish {
      at (place1) async {
        // g'
        val r = do_long_calculation(g);
        at (g) g().value = r; // g''
      }
      do_some_calculation_locally();
    } // end of finish
    Console.OUT.println(o.value);
  }
}
```

- Implicit reference is created in X10 runtime to prohibit the collection
Behavior of the Distributed GC

Collecting the globalized object, and related data

- Weak reference is used to collect related data structures
- Extra inter-place comm. is performed only when a remote ref. is removed

```scala
class GRefExample {
  static class ResultBox { var value:Long=0; }
  public static def main(Rail[String]) {
    val place1 = here.next();
    val o = new ResultBox();
    val g = GlobalRef[ResultBox](o); // g
    finish {
      at (place1) async {
        val r = do_long_calculation(g);
        at (g) g().value = r; // g''
      }
      do_some_calculation_locally();
    } // end of finish
    Console.OUT.println(o.value);
  }
}
```
Summary – Distributed GC

- Distributed GC can collect remotely-referenced objects in Managed X10, which runs on multiple JVMs
  - No modification to the underlying JVMs
- Globalized Object Tracker (GOT) centralizes the control of a globalized (remotely-referenced) object
  - Weak references are used to detect object deletion
  - Strong (normal) references are used to disable collection
Resilient X10

* 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.
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 by GlobalRef
  - 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
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 $\pi$ 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 $\pi$ 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

Assign the work only to live nodes

```scala
class ResilientKMeans {
  public static def main(args:Rail[String]) {
    val iter = 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 */
  }
```

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

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] { ... {
  public static def make[T](dist:Dist, init:(Point)=>T) : DistArray[T];
  public static def make[T](pt:Point){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
Execution Example – HeatTransfer (10x10)

```plaintext
$ 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
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

X10_RESILIENT_STORE_MODE=0
X10_RESILIENT_STORE_VERBOSE=0

--- Iteration: 1
delta=0.25

--- Iteration: 2
delta=0.125

--- Iteration: 10
delta=0.023305892944336
Create a snapshot at iteration 10

--- 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

--- Iteration: 39 <<< Place 2 was killed

--- 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

--- 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

--- Iteration: 87
delta=8.5E-4

--- Iteration: 194
delta=9.7E-6
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 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
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 0.566 0.463 0.385 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 0.212 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 0.064 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 0.061 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
```
(5) Resilient PlaceLocalHandle

- An extended PlaceLocalHandle which supports snapshot and restoration

```
package x10.resilient.lang;
public class PlaceLocalHandle<T> ... {
    public static def make[T](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

Interface overview
- Normal PLH interfaces, plus the followings:
  - snapshot()
  - Dump the data of each place into the Resilient Storage
  - restore(newPg)
  - Reconstruct the PLH over the new PlaceGroup, and restore the snapshot
(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

Added resiliency to MapReduce Engine “M3R Lite”

- When a place is dead during an iterative map-reduce execution, next iteration will be done excluding the dead place.
- Job interface is unchanged
- Some methods are added to the Engine
  - `numLivePlaces()` number of places used for this iteration
  - `placeIndex(Place)` index of specified place
  - `iterationNumber()` current iteration number
  - `iterationFailed()` is last iteration failed?

- Jobs implemented
  - KMeans, HMMTrainer, SparseMatVecMult

```scala
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;
}
```

https://svn.code.sf.net/p/x10/code/applications/trunk/m3rlite/src/com/ibm/m3rlite/ResilientEngine.x10
Get cluster coords
Calculate my part
(e.g. 0~6665)
and cache the data
(Point ID, Coords)

For each point,
determine the nearest cluster
(Cluster ID, Point's Coords)

Caculate new cluster coords
(Cluster ID, New Coords)

Put the new cluster coords
to ResilientStore

if (!iterationFailed)
update clusters from ResStore

if (converged) exit

Points data

Points 0~9999

Points 10000~19999

Tentative clustering results

New cluster 0

New cluster 1

New cluster 2

New cluster 3

ResilientStore

source

mapper

partition

reducer

sink

stop

(8) KMeans Job on Resilient M3R Lite
KMeans Job on Resilient M3R Lite (Skeleton)

    private val master = GlobalRef(this); // master instance
    private var engine:ResilientEngine[Long, Coords, Long, Coords, Long, Coords];
    transient var clusters:Rail[Coords]; // latest cluster coords
    transient var myData:Rail[Coords]; // part of N points for this place ((startIndex,endIndex))
    transient var startIndex:Long=-1, endIndex:Long=-1; // range of myData
    // ResilientStore to store N points of data and tentative cluster values
    val rs_data = ResilientStoreForApp.make[Long/*start ID*/, Rail[Coords/*10000 coords*/]]();
    val rs_clusters = ResilientStoreForApp.make[Long/*cluster ID*/, Coords/*cluster coords*/]() {

            val job = new ResilientKMeansM3R_2(N, NC, ND, Data);
            val engine = new ResilientEngine(job); job.engine = engine;
            engine.run(); return job.clusters;
        }

        // Create a "master" job instance
        public def this(N:Long, NC:Long, ND:Long, Data:Rail[Coords]) {
            for (var i:Long = 0; i < N; i += 10000) {
                Rail.copy(Data, i, chunk, 0, 10000); // put data into ResilientStore (per 10000 points)
            } // put data into ResilientStore (one entry per 10000 points)
        }
    } // Create a "master" job instance
} // Create a "master" job instance

Points used by each place is cached into myData
ResilientStore is used to store points and temporary clustering results
N: Number of points NC: Num of clusters ND: Dimension Data: Points data

© 2015 IBM Corporation
// Return an iterator that generates (data ID, coordinates) for the place
public def source() {
  clusters = at (master) master().clusters; // set the latest cluster info into local job instance
  val placeIndex = engine.placeIndex(here), numLivePlaces = engine.numLivePlaces();
  val s = placeIndex*N/numLivePlaces, e = (placeIndex+1)*N/numLivePlaces;
  if (s != startIndex || e != endIndex) { // get data from ResilientStore
    << read myData from ResilientStore by rs_data.get(...) >>
    startIndex = s; endIndex = e;
  }
  return new Iterable[Pair[Long,Coords]]() { ... }; // return an iterator over myData
}

// Map (data ID, coordinates) to (nearest cluster ID, coordinates of the data)
public def mapper(k:Long, v:Coords, msink:(Long,Coords)=>void) {
  var dmin:Double = Double.MAX_VALUE, idmin:Long = -1;
  for (var i:Long = 0; i < NC; i++) { // find the nearest cluster
    var d:Double = << calculate distance between v and clusters(i) >>;
    if (d < dmin) { dmin = d; idmin = i; }
  }
  msink(idmin, v); // send (nearest cluster ID, coordinates)
}

// Partition mapped data just by cluster ID
public def partition(k:Long) = k;
// Reduce (cluster ID, coordinates list) into (cluster ID, new coordinates of the cluster)
public def reducer(a:Long, b:Iterable[Coords], output:ArrayList[Pair<Long,Coords]]) {
    var pos:Coords = new Coords(ND, 0.0), c:Long = 0;
    for (v in b) { c++;
        for (var j:Long = 0; j < ND; j++) pos(j) += v(j); }
    for (var j:Long = 0; j < ND; j++) pos(j) /= c; // calculate the center
    output.add(Pair(a, pos));
}

// Store the new cluster values (cluster ID, new coordinates of the cluster)
public def sink(s:Iterable[Pair<Long,Coords>]) {
    for (kv in s) rs_clusters.put(kv.first, kv.second); // put the cluster info into ResilientStore
}

// Check to stop the iteration (called only for the master job)
public def stop() {
    if (engine.iterationFailed()) return false; // some livePlace died in the last iteration
    // iteration succeeded without place death, update the cluster values and calculate diff
    var diff:Double = 0.0; // diff from the old clusters
    for (var i:Long = 0; i < NC; i++) {
        val v = rs_clusters.get(i) as Coords; // get the new cluster data from ResilientStore
        diff += << calculate delta between v and clusters(i) >>;
        clusters(i) = v;
    }
    return (diff < Epsilon); // true if converged
}
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 – Resilient X10

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
Summary of the Talk
Summary of the Talk

Introduced a Resilient Distributed Programming Language X10

- **X10 at a Glance**
  - APGAS programming model

- **Distributed processing with X10**
  - Primitives: **at, async, finish**
  - Data structures: DistArray, PlaceLocalHandle, GlobalRef

- **Distributed GC**
  - Utilization of weak reference

- **Fault Tolerance Support by Resilient X10**
  - Resilient programming patterns to use DeadPlaceException
  - Resilient MapReduce Engine

→ Find more at [http://x10-lang.org/](http://x10-lang.org/) !!