Programming Language X10 on Multiple JVMs

2013/05/15
Kiyokuni (KIYO) Kawachiya and Mikio Takeuchi
IBM Research - Tokyo
This Talk Focuses on “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
"Hello World" in X10

**X10 Program**

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

**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
```

For C++ back-end:
```
$ x10c++ -o MyHello MyHello.x10
$ X10_NPLACES=4 runx10 MyHello
Hello World from place 3
Hello World from place 0
Hello World from place 1
Hello World from place 2
```

Parallel distributed execution by creating an activity at each place

X10 program is translated into Java (or C+) source code and further compiled

Executed asynchronously on each place
Distributed Execution Model of X10

APGAS: Asynchronous Partitioned Global Address Space

- A global address space is divided into multiple **places** (≡ computer)
  - Place can contain **activities** and data (**objects**, **structs**, **functions**)
  - In Managed X10, each place is implemented by a Java VM
- An object belongs to a place where it is created
  - Object cannot be accessed from other places, but
  - Object can be *[remotely referenced]* from other places, using **GlobalRef**
X10 Characteristics

Java-like syntax, but uses var, val, def, ...

Operators can also be defined

New 1st class data types, structs and functions

Strong type system – type parameters are not erased

Primitive types (Int, Double, ...) are defined as structs

Rich array class

Parallel/distributed processing

Global data access

X10 Characteristics

Java-like syntax, but uses var, val, def, ...

Operators can also be defined

New 1st class data types, structs and functions

Strong type system – type parameters are not erased

Primitive types (Int, Double, ...) are defined as structs

Rich array class

Parallel/distributed processing

Global data access

X10 Characteristics

Java-like syntax, but uses var, val, def, ...

Operators can also be defined

New 1st class data types, structs and functions

Strong type system – type parameters are not erased

Primitive types (Int, Double, ...) are defined as structs

Rich array class

Parallel/distributed processing

Global data access
Compiling X10 to Java (in X10 2.3.0)

Class/field names are basically same

Constructor is separated into Java constructor and field initializer
- to support inlining by front-end

For private instance method and non-virtual call, additional bridge methods are generated
- to support inlining by front-end

X10

1 class B {
2   def w() = this;
3 }
4
class C(p:Int) extends B {
5   val q:Int; var r:Int;
6   def u() {}
7   private def v() {}
8   def w() = super.w();
9   def this(p:Int) {
10      property(p);
11      q = 1;
12   }
13   static val s = Place.MAX_PLACES;
14   static def t() {}
15 }

Java

1 import x10.core.concurrent.AtomicInteger;
2 import x10.lang.ExceptionInInitializer;
3 import x10.lang.Place;
4 import x10.runtime.impl.java.InitDispatcher;
5 public class C extends B {
6   public int p; // property
7   public int q, r; // instance fields
8   // constructor just for allocation
9   public C(System[] $dummy) {
10      super($dummy);
11   }
12   // instance field initializer
13   final public C $$initS(int p) {
14      this.p = p;
15      this.r = 0;
16      this.q = 1;
17      return this;
18   }
19   // 1-phase constructor for Java interop programmers
20   public C(int p) {
21      this((System[]) null); $$initS(p);
22   }
23   public void u() {} // instance methods
24   private void v() {}
25   // bridge for private instance method
26   public static void $$P(C C) { C.v(); }
27   public B w() { return super.w(); }
28   // bridge for non-virtual call
29   public B $$wS() { return super.w(); }
30   public static void t() {} // static method

(some lines are omitted)
Distributed Execution using Multiple JVMs

- Multi-JVM is supported in X10 2.1.2
  - Each X10 “Place” uses its own Java VM
  - Uses common X10RT (in C++) through JNI

```
“x10” command
```

```
Place P
```

```
JVM
```

```
Initiate Places via SSH
```

```
JVM
```

```
Place Q
```

```
x10.runtime.impl.java.Runtime
```

```
Run activity at place P
```

```
Serialize function into byte array
```

```
Send byte array
```

```
JNI (Java->C++)
```

```
x10rt_send_msg
```

```
Communication Physical Layer (e.g. socket, MPI)
```

```
Byte array message
```

```
x10.runtime.impl.java.Runtime
```

```
Invoke function
```

```
Deserialize function from byte array
```

```
Receive byte array from place P
```

```
JNI (Java->C++)
```

```
x10rt_probe
```

```
Communication Physical Layer (e.g. socket, MPI)
```

```
Msg Buffer
```

```
Byte array message
```

```
Network
```

```
Uses backend unique wire format in X10 2.2.0
```
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()…”

```scala
1 class GRefExample {
  2 static class ResultBox { var value:Int = 0; }
  3 public static def main(Array[String]) {
  4   val place1 = here.next();
  5   val o = new ResultBox();
  6   val g = GlobalRef[ResultBox](o);
  7   finish {
  8     at (place1) async { // create an activity at place1
  9       val r = do_long_calculation(g);
 10       at (g) g().value = r; // set the result through GlobalRef
 11     }
 12     do_some_calculation_locally();
 13   } // end of finish
 14   Console.OUT.println(o.value);
 15 } }
```

Figure. Distributed Processing with GlobalRef.

This object *must not be collected* while its GlobalRefs (g) exist in other places.
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
Behavior of the Distributed GC

Creating and using a GlobalRef

```scala
1 class GRefExample {
2   static class ResultBox { var value:Int = 0; }
3   public static def main(Array[String]) {
4     val place1 = here.next();
5     val o = new ResultBox();
6     val g = GlobalRef[ResultBox](o);  // g
7     finish {
8       at (place1) async {  // g'
9         val r = do_long_calculation(g);
10        at (g) g().value = r;  // g''
11       }
12       do_some_calculation_locally();
13     } // end of finish
14     Console.OUT.println(o.value);
15   }
```

Figure. Data Structures for the Distributed GC.
Behavior of the Distributed GC

Collecting the globalized object, and related data

```scala
1 class GRefExample {
2   static class ResultBox { var value: Int = 0; }
3   public static def main(Array[String]) {
4     val place1 = here.next();
5     val o = new ResultBox();
6     val g = GlobalRef[ResultBox](o); // g
7     finish {
8       at (place1) async { // g'
9         val r = do_long_calculation(g);
10        at (g) g().value = r; // g''
11       }
12       do_some_calculation_locally();
13     } // end of finish
14     Console.OUT.println(o.value);
15   }
```

Extra inter-place comm. is performed only when a remote ref. is removed

Figure. Data Structures for the Distributed GC.

- XRJ: X10 Runtime in Java
- GOT: Globalized Object Tracker
- RRT: Remote Reference Tracker
Evaluation: Distributed Fibonacci

- Calculates $n$-th Fibonacci number recursively using multiple places
  - Field `root` is a `GlobalRef` that points to the “root object”
  - Field `v` of the root object is accessed by “at (root) root().v ...”

```
1 class DistFib {
2   static val atomicI = new x10.util.concurrent.AtomicInteger(0);
3   static def getAvailPlace() { // returns next available place
4     val i = at (Place.FIRST_PLACE) atomicI.incrementAndGet();
5     return Place(i % Place.MAX_PLACES);
6   }
7
8   private val root = GlobalRef[DistFib](this); // ref to root obj
9   transient var v:Int; // in-out param, exists only in the root
10  def this(n:Int) { v = n; }
11  def compute() { // global method, can be invoked at any place
12    val n = at (root) root().v; // get the input from the root object
13    if (n < 2) return; // Fib(0)==0, Fib(1)==1
14    val f1 = new DistFib(n-1), f2 = new DistFib(n-2);
15    finish {
16      at (getAvailPlace()) async f1.compute(); // compute remotely
17      f2.compute(); // compute here
18    }
19    val r = f1.v + f2.v; // result = Fib(n-1) + Fib(n-2)
20    at (root) root().v = r; // set the result to the root object
21  }
22
23  public static def main(args:Array[String](1)) {
24    var n:Int = 10; if (args.size > 0) n = Int.parseInt(args(0));
25    val f = new DistFib(n); f.compute(); // this line is measured
26    Console.OUT.println(f.v);
27  }
28}
```

Figure. Distributed Fibonacci.
**Result: Distributed Fibonacci**

Execution times to calculate \( n \)-th Fibonacci numbers using 16 places

- **For very small \( n \)'s, base X10 is faster by about 10%**
  - Because of the small overhead in the Dist. GC version
- **For \( n=25 \), distributed GC version is 13% faster**
  - Because heap is not consumed by uncollectable objects
  - Heap size of Place 0 reaches 468MB in base, but 148MB in Dist. GC

- Logarithmic scale
- Elapsed time in msec (smaller is better)
- Environment: 6-core Xeon X5670 blades (HT disabled)
  - 32GB memory, 40Gbps Infiniband
  - Red Hat Enterprise Server Linux 5.5 x86_64
  - IBM J9 Java VM for Linux x86_64, Version 6.0 SR9
  - Generational GC, -Xms4m –Xmx512m
  - 16 places by starting 4 JVMs on each of 4 blades
Summary of the Talk

Explained implementation details of Managed X10 (X10 on Java VMs)

- Challenges in compiling X10 to Java
  - For performance and functionality

- Distributed GC over Multiple Java VMs
  - No modification to the underlying JVMs
  - Introduced two data structures for monitoring globalized objects and remote references