Integrating Task Parallelism with Actors

By Shams Imam and Vivek Sarkar from Rice University

CCC Seminar Presentation
Otto Bibartiu

Combine Task-Parallelism with the Actor's Paradigm
Parallel Programming Models

- **Library based**
  - Posix Threads
  - MPI

- **Compiler indications**
  - OpenMP

- **Language based**
  - Pig Latin
  - X10
  - Habanero Scala (HS) (Unified Model)
  - Habanero Java (HJ) (Unified Model)
public class Foo{
    public static void main( String[] args)
    {
        System.out.println("T0");
        async big_computation(); // is called asynchronous as a task;
        System.out.println("T1");
    }
    static void big_computation(){
        System.out.println("A1");
    }
}
TP with the Async-Finish-Model (AFM) in HJ

```java
public class Foo{
    public static void main( String[] args)
    {
        System.out.println("T0");
        async big_computation()  // is called asynchronous as a task;

        finish{
            async{
                System.out.println("A2");
            }
            async{
                System.out.println("A3");
                async System.out.println("A4");
                async System.out.println("A5");
            }
        }
        System.out.println("T1");
    }
}

static void big_computation(){
    System.out.println("A1");
}
```

![Diagram showing the flow of execution]
public class Foo{
    public static void main( String[] args)
    {
        var i = 0
        System.out.println("T0");
        async one_big_computation() // is called asynchronous as a task;
        finish{
            async{
                System.out.println("A2");
            }
            async{
                System.out.println("A3");
                async System.out.println("A4");
                async System.out.println("A5");
            }
            finish{
                async i++;
                async i--;
            }
            System.out.println("A6 i="+i);
        }
        System.out.println("T1");
    }
    static void one_big_computation(){
        System.out.println("A1");
    }
}
Actor Model

- A universal modular ACTOR formalism for artificial intelligence '73
  - Authors: Carl Hewitt, Peter Bishop, Richard Steiger
- Actors is a processes which communicate only via messages
  - Send / receive
  - Processes only one message at the time
  - Change of local state
  - Create new Actors

Actor in HS

```scala
object Boo extends HabaneroApp {
  val printActor = new PrintActor()
  printActor.start()
  printActor ! "Hallo World"
  printActor ! True
}

class PrintActor extends HabaneroReactor{
  // Local State
  def behavior () = {
    case msg: Boolean => exit()
    case msg: String => println(msg)
  }
}
```

- How to detect when an Actor has finished?
Actors in Scala

```scala
object ScalaActorApp extends App {
  val latch = new CountDownLatch ( 1 )
  val actor = new PrintActor(latch)
  actor ! "Hello World"
  actor ! True
  latch.await ()
  println("Actor terminated")
}

class PrintActor(latch: CountDownLatch) extends Actor{
  def act() = {
    case msg: Boolean => {
      // Lots of computation
      latch.countDown()
      exit()
    }
    case msg: String => println(msg)
  }
}
```

- What about Actors calling Actors?
  - Is getting difficult when joining child actors
Actors in HS

• AFM + Actors = Unified Model
• (Child) Actors inherit the immediate enclosing Finish
• Enclosing Finish of actor is the Finish where actor.start() was performed

```scala
object ScalaActorApp extends App {
  val actor = new PrintActor
  finish{
    actor.start()
    actor ! "Hello World"
    actor ! True
  }
  println("Actor terminated")
}

class PrintActor extends HabaneroReactor{
  def behavior() = {
    case msg: Boolean => exit()
    case msg: String => println(msg)
  }
}
```
object Goo extends HabaneroApp {

    val printActor = new PrintActor()
    printActor.start()
    printActor ! "Hallo World"
}

class PrintActor extends HabaneroReactor{

    def behavior () = {
        case msg: String =>{
            finish{
                async stmt_1
                async stmt_1
                // parallel computation
                // with many asyncs
            }
            // but ...
            async{
                // violation of the one message invariant
            }
        }
    }
}
object Goo extends HabaneroApp {
    val printActor = new PrintActor()
    printActor.start()
    printActor ! "Hallo World"
}

class PrintActor extends HabaneroReactor{
    def behavior () = {
        case msg: String =>{
            finish{
                async stmt_1
                async stmt_1
                // parallel computation
                // with many asyncs
            }
        }
        pause() // no message will be processed
        // new messages are still being received and kept in mailbox
        async{
            // do critical computation
            resume() // we are save now! allow actor to process messages
            // continue uncritical computation
        }
    }
}
Futures in HJ

Java version

```java
Callable<ImageData> c1 = new Callable<ImageData>() {
    public ImageData call() { return imageInfo.downloadImage(1); }
};
FutureTask<Object> ft1 = new FutureTask<Object>(c1);
new Thread(ft1).start();
Callable<ImageData> c2 = new Callable<ImageData>() {
    public ImageData call() { return imageInfo.downloadImage(2); }
};
FutureTask<Object> ft2 = new FutureTask<Object>(c2);
new Thread(ft2).start();
renderImage(ft1.get());
renderImage(ft2.get());
```

HJ version

```java
future<ImageData> ft1 = async<ImageData>{return imageInfo.downloadImage(1);};
future<ImageData> ft2 = async<ImageData>{return imageInfo.downloadImage(2);};
renderImage(ft1.get());
renderImage(ft2.get());
```
Data Driven Futures in HS

```scala
object Foo extends HabaneroApp {
  val (ddf_1, ddf_2) = (ddf(), ddf())

  async{
    //long computation
    dff_1.put(4)
  }
  async{
    //small computation
    dff_2.put(6)
  }

  asyncAwait(ddf_1, ddf_2){
    System.out.println(ddf_1.get() + ddf_2.get());
  }
}
```

asyncAwait can have a list of DDFs. It will wait until all values of all DDFs are available.