SKANDIUM

Distributed systems: paradigms and models

Monday 18th November, 2013

Tiziano De Matteis

Email: dematteis<at>di.unipi.it

OUTLINE

- Skandium library overview
- Using Skandium Skeletons
- Sample applications and demo

HISTORY

Skandium is a **skeleton framework** provided as a library, conceived by Mario Leyton (INRIA, University of Chile and, currently, Google).

It is a complete Java reimplementation of **Calcium** (part of the **ProActive Middleware**), supporting only multi-/many-core architectures exploiting Java thread facilities.

Its implementation is based on a Macro Data Flow engine (but uses *evaluation stacks*).

URLs: **Snapshot of the official site**
(http://backus.di.unipi.it/∼marcod/SkandiumClone/skandium.niclabs.cl/)
Github (need to be patched)
(https://github.com/mleyton/Skandium)

USING SKANDIUM'S SKELETONS

Skeleton programs have a well defined structure, that is usually represented by a **skeleton tree**.

The skeleton tree has nodes representing algorithmic skeletons and leaves representing sequential portions of code (the business logic code).

The same concepts can be applied in writing a Skandium program.

USING SKANDIUM'S SKELETONS

In Skandium, skeletons are provided as a Java library and the programmer can nest **task** and **data parallel** skeletons in the following way:

$$\begin{aligned}\triangle ::= &\ \text{seq}(f_e) \mid \text{farm}(\triangle) \mid \text{pipe}(\triangle_1, \triangle_2) \mid \text{while}(f_c, \triangle) \mid \\ &\ \text{if}(f_c, \triangle_{true}, \triangle_{false}) \mid \text{for}(i, \triangle) \mid \text{map}(f_s, \triangle, f_m) \mid \\ &\ \text{fork}(f_s, \{\triangle_i\}, f_m) \mid \text{d\&c}(f_c, f_s, \triangle, f_m)\end{aligned}$$

USING SKANDIUM'S SKELETONS

In Skandium, skeletons are provided as a Java library and the programmer can nest **task** and **data parallel** skeletons in the following way:

$$\begin{aligned}\triangle ::= &\ \text{seq}(f_e) \mid \text{farm}(\triangle) \mid \text{pipe}(\triangle_1, \triangle_2) \mid \text{while}(f_c, \triangle) \mid \\ &\ \text{if}(f_c, \triangle_{true}, \triangle_{false}) \mid \text{for}(i, \triangle) \mid \text{map}(f_s, \triangle, f_m) \mid \\ &\ \text{fork}(f_s, \{\triangle_i\}, f_m) \mid \text{d\&c}(f_c, f_s, \triangle, f_m)\end{aligned}$$

f_* are sequential blocks (called *muscles*) provided by the programmer:

- f_e is an execution block;
- f_c evaluation of a condition;
- f_s split of data;
- f_m merge of results.

USING SKANDIUM'S SKELETONS

In Skandium, skeletons are provided as a Java library and the programmer can nest **task** and **data parallel** skeletons in the following way:

$$\begin{aligned}\triangle ::= &\ \mathrm{seq}(f_e) \mid \mathrm{farm}(\triangle) \mid \mathrm{pipe}(\triangle_1, \triangle_2) \mid \mathrm{while}(f_c, \triangle) \mid \\ &\ \mathrm{if}(f_c, \triangle_{true}, \triangle_{false}) \mid \mathrm{for}(i, \triangle) \mid \mathrm{map}(f_s, \triangle, f_m) \mid \\ &\ \mathrm{fork}(f_s, \{\triangle_i\}, f_m) \mid \mathrm{d\&c}(f_c, f_s, \triangle, f_m)\end{aligned}$$

We will review some of them: pipeline, farm and map.

STRUCTURE OF A SKANDIUM PROGRAM

Let's see what is the general structure of a generic Skandium program:

1. Definition of Skandium environment and skeleton tree. Each skeleton:
   - is defined by extending the abstract class `AbstractSkeleton`, that implements its basic functionalities;
   - has two constructors in order to allow the compositionality (one to define a standalone/leaf skeleton, the other for nesting).
2. Acquisition of the input stream and injection of the input elements;
3. Acquisition of the results, by means of `Futures`.

FUTURES

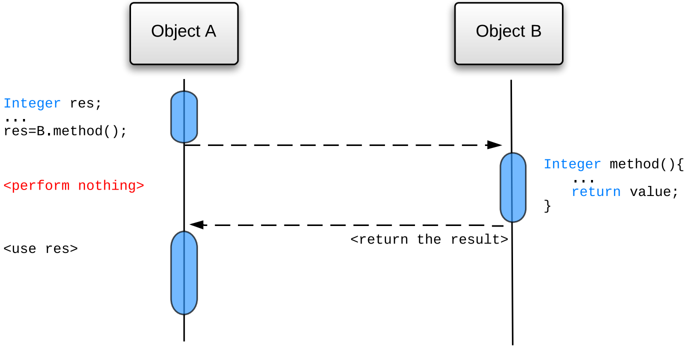
Futures are natively supported by Java from release 5.0 and are used to represent the result of an asynchronous computation.

Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

FUTURES

Futures are natively supported by Java from release 5.0 and are used to represent the result of an asynchronous computation.

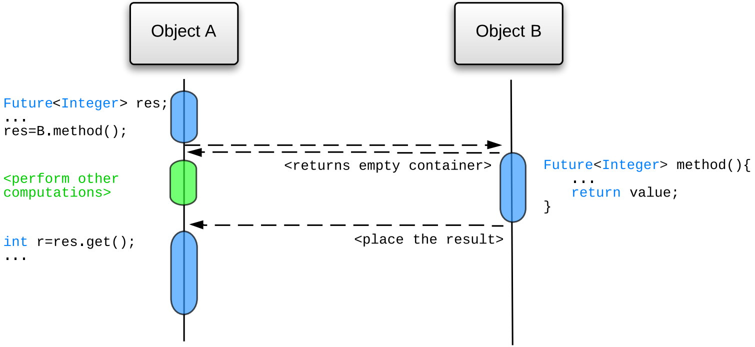
Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.



FUTURES

Futures are natively supported by Java from release 5.0 and are used to represent the result of an asynchronous computation.

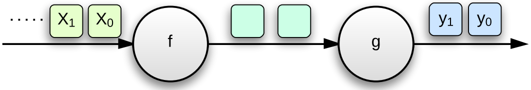
Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.



PIPELINE

Used to represent the composition of functions:

$$y_i = g(f(x_i))$$



Class definition:

```
public class Pipe<P,R> extends AbstractSkeleton<P,R>
```

Constructors:

```
public Pipe(Skeleton<P,X> stage1, Skeleton<X,R> stage2)
public Pipe(Execute<P,X> stage1,Execute<X,R> stage2)
```

PIPELINE: EXAMPLE

Let's define a two stage pipeline that operates on a stream of integer x_i. The result of the computation is $y_i = (x_i + 1)^2$

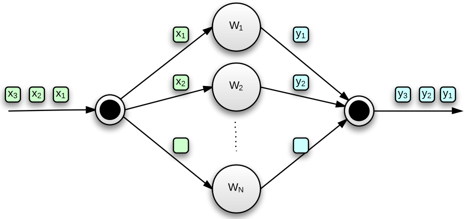
```
// First stage
class Incr implements Execute<Integer, Integer> {
   public Integer execute(Integer arg0) {
      return ++arg0;
   }
}

// Second stage
class Square implements Execute<Integer, Integer> {
   public Integer execute(Integer arg0) {
      return arg0 * arg0;
   }
}
```

PIPELINE: EXAMPLE

```java
public class SimplePipeline {
   public static void main(String[] args) {
      int size = 4;
      Vector<Future<Integer>> futures = new Vector<Future<Integer>>();


      // create the Skandium environment
      Skandium skandium = new Skandium();

      // define the pipeline skeleton
      Skeleton<Integer, Integer> pipeline = new Pipe<Integer, Integer>(new Incr(),
            new Square());


      // get the input stream
      Stream<Integer, Integer> stream = skandium.newStream(pipeline);

      // pass the input
      for (int i = 0; i < size; i++)
         futures.add(stream.input(i));


      // get the results and print
      for (Future<Integer> future : futures)
            System.out.println(future.get());

      // shutdown the Skandium environment
      skandium.shutdown();
   }
}
```

FARM

The Farm paradigm is based on the **replication** of a function. Each Worker computes the same application code on different input tasks:



Class definition:

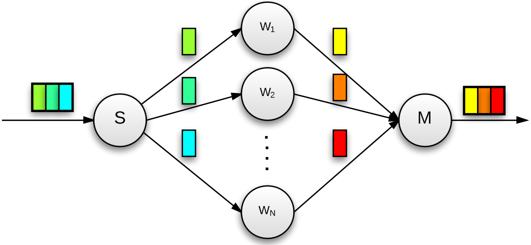
```
public class Farm<P,R> extends AbstractSkeleton<P,R>
```

Constructors:

```
public Farm(Skeleton<P,R> skeleton)
public Farm(Execute<P,R> execute)
```

MAP

In the Map paradigm, the computation is performed by multiple Workers that elaborate on **partitions** of the input task. We will have a first stage in which the input data is **split**, the calculus and the a final stage in which partial results are **merged** together.



MAP (2)

Class/Interface definitions:

```
public class Map<P,R> extends AbstractSkeleton<P,R>
public interface Split<P,R> extends Muscle<P,R>
public interface Merge<P,R> extends Muscle<P,R>
```

Constructors:

```
public Map(Split<P,X> split, Execute<X,Y> execute, Merge
    <Y,R> merge)
public Map(Split<P,X> split, Skeleton<X,Y> skeleton,
    Merge<Y,R> merge)
```

QUESTIONS?