COMP 322: Fundamentals of Parallel Programming

Lecture 10: Java's ForkJoin Library

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

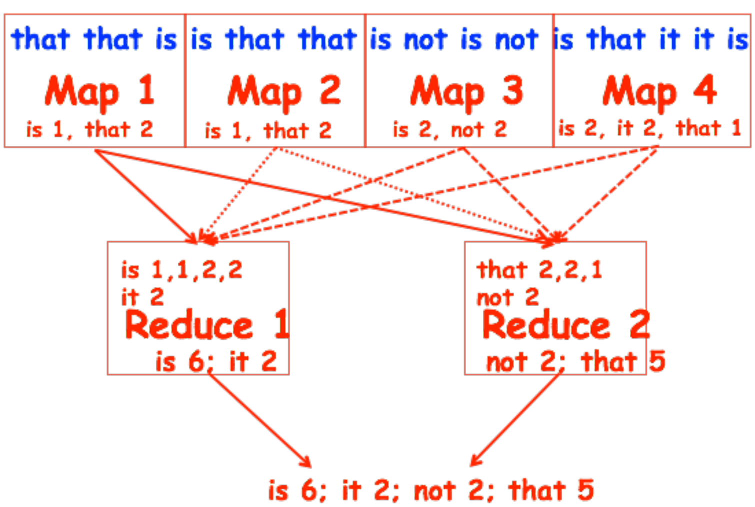
Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

Worksheet #9: Analysis of Map Reduce Example

Analyze the total WORK and CPL for the Map reduce example:

- Assume that each Map step has WORK = number of input words, and CPL=1
- Assume that each Reduce step has WORK = number of input word-count pairs, and CPL = log2(# occurrences for input word with largest # pairs)



WORK/CPL for all Map steps:

- WORK = 15
- CPL = 1

WORK/CPL for Reduce 1 step:

- WORK = 5
- CPL = log2(4) = 2

WORK/CPL for Reduce 2 step:

- WORK = 4
- CPL = log2(3) = 1.58

Total WORK and CPL

- WORK = 15+5+4 = 24
- CPL = 1 + 2 = 3

Updating all Elements in an Array

- Suppose we have a large array *a* of integers
- We wish to update each element of this array:
  - `a[i] = a[i] / (i + 1)`
- How would we write this as a parallel program using `async` and `finish`?

Recursive Decomposition

```
solve(problem)
    if problem smaller than threshold
        solveDirectly(problem)
    else
    in parallel:
        l = solve(left-half)
        r = solve(right-half)
    combine(l, r)
```

- In general, can create more than 2 sub-problems
- `combine` then needs to handle all the sub-problems

Update using `async` and `finish`

```
sequentialUpdate(a, lo, hi)
    for (i = lo; i < hi; i++)
        a[i] = a[i] / (i + 1)

parallelUpdate(a, lo, hi)
    if (hi - lo) < THRESHOLD
        sequentialUpdate(a, lo, hi)
    else
        mid = (lo + hi) / 2
       finish
           async parallelUpdate(a, lo, mid)
           async parallelUpdate(a, mid, hi)
```


Using Java's Fork/Join Library

- Today, we will look at popular library for task parallelism available since Java 7
- We can perform recursive subdivision using the Fork/Join libraries provided in the JDK as follows:

```
public abstract class RecursiveAction extends ForkJoinTask<Void> {
    protected abstract void compute();
    …
}
public abstract class RecursiveTask<V> extends ForkJoinTask<V> {
    protected abstract V compute();
    …
}
```

Implementing a subclass of RecursiveAction

```
class DivideTask extends RecursiveAction {
  static final int THRESHOLD = 5;
  final long[] array;
  final int lo, hi;

  DivideTask(long[] array, int lo, int hi) {
    this.array = array;
    this.lo = lo;
    this.hi = hi;
  }
  protected void compute() {…} // next slide
}
```


Implementing compute()

```
protected void compute() {
  if (hi - lo < THRESHOLD) {
    for (int i = lo; i < hi; ++i)
      array[i] = array[i] / (i + 1);
  } else {
    int mid = (lo + hi) >>> 1;
    invokeAll(new DivideTask(array, lo, mid),
              new DivideTask(array, mid, hi));
  }
}
```

invokeAll

- Defined in `java.util.concurrent.ForkJoinTask` (parent class for `RecursiveAction`)

```
class ForkJoinTask<V> extends Object
    implements Serializable, Future<V> {

    static void invokeAll(ForkJoinTask<?>… tasks)
    static void invokeAll(Collection<T> tasks)
    …
}
```

- There are many helper methods in `ForkJoinTask`; we highlight just a few
- See the Java API for more (Google is your friend)

ForkJoinTask<V>

- Similar to a finish block enclosing a collection of asyncs
- Other Fork/Join methods in superclass ForkJoinTask<V>

```
class ForkJoinTask<V> extends Object
    implements Serializable, Future<V>
{
    ForkJoinTask<V> fork()   // asynchronously executes
    V join()        // returns result when execution completes
    V invoke()      // forks, joins, returns result
    …
}
```

ForkJoinTasks and Futures

- ForkJoinTasks implement the Future interface
- Acts very much like HJLib futures

```
interface Future<V> {
    V get()
    V get(long timeout, TimeUnit unit)
    boolean cancel(boolean interruptIfRunning)
    boolean isCancelled()
    boolean isDone()
}
```


ForkJoinTasks and Futures

- Because ForkJoinTasks are Futures, they are the values returned from `fork()`
- We can obtain the result of a ForkJoinTask using `join()` or `get()`
- When calling `invoke` or `invokeAll`, we never get a handle on the future explicitly
  - Similar to `finish`/`async` blocks in HJLib

Recursive Array Sum using HJlib Futures

```java
protected double computeSum(
        final double[] xArray, final int start, final int end)
        throws SuspendableException {

    if (end - start < THRESHOLD) {

        // sequential threshold cutoff
        return seqArraySum(xArray, start, end);

     } else {
         int mid = (end + start) / 2;

         HjFuture<Double> leftFuture = future(() -> {
             return computeSum(xArray, start, mid);
         });
         HjFuture<Double> rightFuture = future(() -> {
             return computeSum(xArray, mid, end);
         });
         return leftFuture.get() + rightFuture.get();
} }
```


Recursive Array Sum using ForkJoinTasks

```java
protected static class ArraySumForkJoinTask
        extends RecursiveTask<Double> {
    private final double[] xArray;
    private final int start;
    private final int end;

    protected Double compute() {
        if (end - start < THRESHOLD) {
            // sequential threshold cutoff
            return seqArraySum(xArray, start, end);
        } else {
            int mid = (end + start) / 2;
            ArraySumForkJoinTask taskLeft =
                    new ArraySumForkJoinTask(xArray, start, mid);
            ArraySumForkJoinTask taskRight =
                    new ArraySumForkJoinTask(xArray, mid, end);

            // taskLeft.fork(); taskRight.fork();
            invokeAll(taskLeft, taskRight);

            return taskLeft.join() + taskRight.join();
} } }
```

Recursive Array Sum using ForkJoinTasks Optimized

```java
protected static class ArraySumForkJoinTask
        extends RecursiveTask<Double> {
    ...

    protected Double compute() {
        if (end - start < THRESHOLD) {
            // sequential threshold cutoff
            return seqArraySum(xArray, start, end);
        } else {
            final int mid = (end + start) / 2;
             final ArraySumForkJoinTask taskLeft =
                         new ArraySumForkJoinTask(xArray, start, mid);
             final ArraySumForkJoinTask taskRight =
                         new ArraySumForkJoinTask(xArray, mid, end);

             taskRight.fork();
             return taskLeft.compute() + taskRight.join();

             // What is wrong with the code below?
             // taskLeft.fork();
             // return taskLeft.join() + taskRight.compute();
} } }
```


ForkJoinPools

- ForkJoinTasks are executed by the threads in a ForkJoinPool
- By default, contains a number of threads equal to the number of available processors (`java.lang.Runtime.availableProcessors()`)
- You can create your own `ForkJoinPools`
  - But you hardly ever need to

```java
class ForkJoinPool {
    static ForkJoinPool commonPool()
    …
}
```

- The common pool is used by any `ForkJoinTask` not explicitly *submitted* to a specific pool