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

```python
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

1. sequentialUpdate(a, lo, hi)
2. for (i = lo; i < hi; i++)
3.     a[i] = a[i] / (i + 1)
4.
5. parallelUpdate(a, lo, hi)
6. if (hi - lo) < THRESHOLD
7.     sequentialUpdate(a, lo, hi)
8. else
9.     mid = (lo + hi) / 2
10. finish
11. async parallelUpdate(a, lo, mid)
12. 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:

```java
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

1. class DivideTask extends RecursiveAction {
2.     static final int THRESHOLD = 5;
3.     final long[] array;
4.     final int lo, hi;
5.     
6.     DivideTask(long[] array, int lo, int hi) {
7.         this.array = array;
8.         this.lo = lo;
9.         this.hi = hi;
10.     }
11.     protected void compute() {...} // next slide
12. }

Implementing compute()

1.     protected void compute() {
2.         if (hi - lo < THRESHOLD) {
3.             for (int i = lo; i < hi; ++i)
4.                 array[i] = array[i] / (i + 1);
5.         } else {
6.             int mid = (lo + hi) >>> 1;
7.             invokeAll(new DivideTask(array, lo, mid),
8.                 new DivideTask(array, mid, hi));
9.         }
10.     }
invokeAll

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

```java
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>`

```java
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

```java
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

1. protected double computeSum(
2.     final double[] xArray, final int start, final int end)
3.     throws SuspendableException {
4.         if (end - start < THRESHOLD) {
5.             // sequential threshold cutoff
6.             return seqArraySum(xArray, start, end);
7.         } else {
8.             int mid = (end + start) / 2;
9.             HjFuture<Double> leftFuture = future(() -> {
10.                 return computeSum(xArray, start, mid);
11.             });
12.             HjFuture<Double> rightFuture = future(() -> {
13.                 return computeSum(xArray, mid, end);
14.             });
15.             return leftFuture.get() + rightFuture.get();
16.         }
17.     }

Recursive Array Sum using ForkJoinTasks

1. protected static class ArraySumForkJoinTask
2.     extends RecursiveTask<Double> {
3.         private final double[] xArray;
4.         private final int start;
5.         private final int end;
6.         
7.         protected Double compute() {
8.             if (end - start < THRESHOLD) {
9.                 // sequential threshold cutoff
10.                 return seqArraySum(xArray, start, end);
11.             } else {
12.                 int mid = (end + start) / 2;
13.                 ArraySumForkJoinTask taskLeft =
14.                     new ArraySumForkJoinTask(xArray, start, mid);
15.                 ArraySumForkJoinTask taskRight =
16.                     new ArraySumForkJoinTask(xArray, mid, end);
17.                 invokeAll(taskLeft, taskRight);
18.                 return taskLeft.join() + taskRight.join();
19.             }
20.         }

21.     }
22. }
Recursive Array Sum using ForkJoinTasks Optimized

```java
1. protected static class ArraySumForkJoinTask
2.     extends RecursiveTask<Double> {
    ... 
3. 
4.     protected Double compute() {
5.         if (end - start < THRESHOLD) {
6.             // sequential threshold cutoff
7.             return seqArraySum(xArray, start, end);
8.         } else {
9.             final int mid = (end + start) / 2;
10.            final ArraySumForkJoinTask taskLeft =
11.                new ArraySumForkJoinTask(xArray, start, mid);
12.            final ArraySumForkJoinTask taskRight =
13.                new ArraySumForkJoinTask(xArray, mid, end);
14.            taskRight.fork();
15.            return taskLeft.compute() + taskRight.join();
16.         } 
17.     }
```

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