Parallel Programming with Fork/Join

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or
custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Basics**
  - Concurrent vs. parallel programming
  - General fork/join strategy
  - Making code to time operations
  - Summing an array: simple approach
  - Preview: array sum with Java 8 streams
- **Advanced**
  - Defining a reusable parallel array processor
  - Summing an array
  - Finding minimum of complex calculations
  - Generating large prime numbers



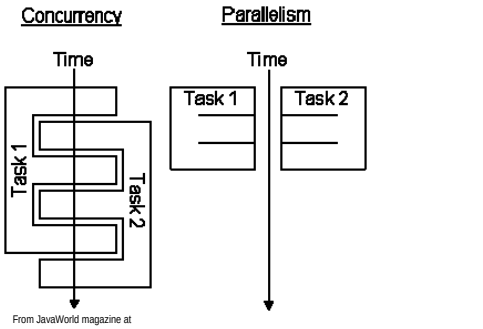




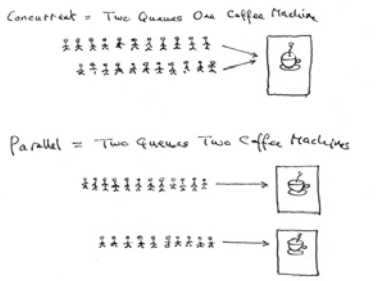
Overview

Concurrent vs. Parallel Programming

- **Concurrent**
  - Tasks that overlap in time
    - The system might run them in parallel on multiple processors, or might switch back and forth among them on the same processor
- **Parallel**
  - Tasks that run at the same time on different processors



From JavaWorld magazine at http://www.javaworld.com/article/2076774/java-concurrency/programming-java-threads-in-the-real-world--part-1.html



From Joe Armstrong at http://joearms.github.io/2013/04/05/concurrent-and-parallel-programming.html

Java Threads (Concurrent) vs. Fork/Join Framework (Parallel)

- **Using threads**
  - When task is relatively large and self-contained
  - Usually when you are waiting for something, so would benefit even if there is only one processor
  - Covered in this lecture
    - Needed even in Java 8 where you have parallel streams
- **Using fork/join or parallel streams**
  - When task starts large but can be broken up repeatedly into smaller pieces, then combined for final result.
  - No benefit if there is only one processor
  - Covered in next lecture
    - Also, parallel version of Streams, covered in separate lectures, uses fork/join framework under the hood, can handle the majority of situations where you want to use fork/join, and is dramatically simpler than using fork/join explicitly. Most Java 8 developers can skip learning fork/join and concentrate on parallel streams instead.

Best Practices: Both Fork/Join and Parallel Streams

- **Check that you get the same answer**
  - Verify that sequential and parallel versions yield the same (or close enough to the same) results. This can be harder than you think when dealing with doubles.
- **Check that the parallel version is faster**
  - Or, at the least, no slower. Test in real-life environment.
  - If running on app server, this analysis might be harder than it looks. Servers automatically handle requests concurrently, and if you have heavy server load and many of those requests use parallel streams, all the cores are likely to *already* be in use, and parallel execution might have little or no speedup.
    - Another way of saying this is that if the CPU of your server is already consistently maxed out, then parallel streams will not benefit you (and could even slow you down).

Summary of Fork/Join Coding Strategy

Using Fork/Join: Summary

- **Make a subclass of RecursiveTask<T>**
  - Pass any needed arguments to constructor of class
- **Put real work in "compute" method**
  - If task is small, calculate and return answer directly
  - If task is large, break into two pieces, each of which is a new instance of your subclass
    - Call leftPiece.fork() to start the left piece running in parallel
    - Call rightPiece.compute() to start the right piece directly
    - Call leftPiece.join() to wait for the left piece to finish
    - Combine the results of the two pieces and return that
- **Make an instance of ForkJoinPool**
  - Call pool.invoke on instance of your class

Using Fork/Join: Pseudo-Code (Recursive Task)

```
public class MyTask extends RecursiveTask<ResultType> {
  ... // Instance vars that have data and determine size

  public MyTask(...) { ... }  // Set instance vars

  @Override
  protected ResultType compute() {
    if (size < someCutoff) {
      return(directlyCalculatedResult(...));
    } else {
      MyTask leftTask = new MyTask(...);
      MyTask rightTask = new MyTask(...);
      leftTask.fork();  // Run on separate processor
      ResultType rightResult = rightTask.compute();// Run now
      ResultType leftResult = leftTask.join();   // Wait
      return(someCombinationOf(leftResult, rightResult));
    }
  }}
```

Using Fork/Join: Pseudo-Code (Top-Level Driver)

```
private static final ForkJoinPool FORK_JOIN_POOL =
  new ForkJoinPool();

public ResultType parallelCompute(...) {
  ResultType result = FORK_JOIN_POOL.invoke(new MyTask(...));
  return(result);
}
```








Aside: Reusable Timing Code

Collecting Performance Stats

- **Problem**
  - You cannot simply make a timeOp method that you call like this:
    ```
    Utils.timeOp(someMethodCall(…));
    ```
    - Why not?
- **Solution 1 (now)**
  - Define an Op interface
  - Use anonymous inner class to pass instance of Op to the timeOp method
- **Solution 2 (later)**
  - Define an Op interface
  - Use lambda expression to pass "function" to timeOp
    - Technically still an instance of Op under the hood

TimingUtils.timeOp

- **Timing results are needed pervasively**
  - In all fork/join examples, you need to compare speed of the sequential version to the parallel version
- **Don't want to repeat code, so**
  - Define an interface (Op) with a runOp method that should contain the code to be timed and then return a descriptive string.
  - Implement Op in concrete classes that define a runOp method that performs the work that should be timed
  - Make reusable method (TimingUtils.timeOp) to run code, print descriptive string, and show timing results

Op Interface

```
public interface Op {
  String runOp();
}
```


Timing Method

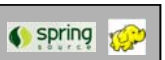
```
public class TimingUtils {
  private static final double ONE_BILLION = 1_000_000_000;

  public static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    String resultMessage = operation.runOp();
    long endTime = System.nanoTime();
    System.out.println(resultMessage);
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf(Elapsed time: %.3f seconds.%n",
                      elapsedSeconds);
  }
}
```

Quick Example of Usage

```
TimingUtils.timeOp(new Op() {
  @Override
  public String runOp() {
    SomeType result = codeToBeTimed(...);
    String message = "Message with %s placeholders for results";
    return(String.format(message, result, ...));
  }
});
```







Summing Large Array of doubles: Version 1

Idea

- **Goal**
  - Sum large array of doubles
- **Steps**
  - Sum it sequentially
  - Sum it in parallel
    - Inside compute, use sequential sum for small sizes
  - Compare results
    - Be sure you get "same" answer both ways
      - But note that reordering addition or multiplication of doubles can sometimes yield slightly different results due to roundoff error. You should define a value where you are willing to accept differences within that delta. Also, in this case, even when results are different, it is not usually clear which is "right", the sequential or parallel version.
    - See which approach is faster
      - We will discuss guidelines for when parallel approach is good, but always time both approaches to confirm

Code: Sequential Sum

```
public class MathUtils {
  public static double arraySum(double[] nums,
                                int lowerIndex, int upperIndex) {
    double sum = 0;
    for(int i=lowerIndex; i<=upperIndex; i++) {
      sum += nums[i];
    }
    return(sum);
  }

  public static double arraySum(double[] nums) {
    return(arraySum(nums, 0, nums.length-1));
  }

  ...
}
```

Code: Parallel Summer (Part 1)

```
public class ParallelArraySummer extends RecursiveTask<Double> {
  private static final int PARALLEL_CUTOFF = 1000;
  private double[] nums;
  private int lowerIndex, upperIndex;

  public ParallelArraySummer(double[] nums,
                             int lowerIndex, int upperIndex) {
    this.nums = nums;
    this.lowerIndex = lowerIndex;
    this.upperIndex = upperIndex;
  }
```


Code: Parallel Summer (Part 2)

```
  @Override
  protected Double compute() {
    int range = upperIndex - lowerIndex;
    if (range <= PARALLEL_CUTOFF) {
      return(MathUtils.arraySum(nums, lowerIndex, upperIndex));
    } else {
      int middleIndex = lowerIndex + range/2;
      ParallelArraySummer leftSummer =
        new ParallelArraySummer(nums, lowerIndex, middleIndex);
      ParallelArraySummer rightSummer =
        new ParallelArraySummer(nums, middleIndex+1, upperIndex);
      leftSummer.fork();
      Double rightSum = rightSummer.compute();
      Double leftSum = leftSummer.join();
      return(leftSum + rightSum);
    }
  }
}
```

I have many more complex examples later, but if you can understand this code, you are ready to write your own fork/join apps. Don't be intimidated!

Code: Parallel Summer (Driver)

```
public class MathUtils {
  private static final ForkJoinPool FORK_JOIN_POOL = new ForkJoinPool();
  ...

  public static Double arraySumParallel(double[] nums) {
    return(FORK_JOIN_POOL.invoke(new ParallelArraySummer(nums,
                                                         0,
                                                         nums.length-1)));
  }
}
```


Helper Method for Building Arrays (Used by Test Routines)

```
public class MathUtils {
  ...
  public static double[] randomNums1(int length) {
    double[] nums = new double[length];
    for(int i=0; i<length; i++) {
      nums[i] = Math.random();
    }
    return(nums);
  }
}
```

Summing Large Array of doubles Version 1: Verification Step

Verifying Results

- **Make sure sequential and parallel versions give same answer**
  - But note that multiplication and addition of doubles is not truly associative, due to roundoff error
    - (a + b) + c is not always equal to a + (b + c) or (a + c) + b
    - If you need *exactly* the same result, then parallel version will not be legal. But, if they differ, it is often hard to say which of the two is more correct.
    - If you need "almost" the same result, you need to define what "almost" is, and then verify that the two answers are within that delta
- **Check that the parallel version is faster**
  - On real-life problem sizes
  - On whatever *n*-processor system you will run it on
  - Under realistic server load if applicable

Verifying Both Approaches Give "Same" Answer

```
public class Tester1 {
  ...
  @Test
  public void testSums() {
    for (int i=1; i<5; i++) {
      int arraySize = (int)Math.pow(10, i);
      double[] nums = MathUtils.randomNums1(arraySize);
      double sum1 = MathUtils.arraySum(nums);
      double sum2 = MathUtils.arraySumParallel(nums);
      assertThat(sum1, is(closeTo(sum2, 0.000001)));
    }
  }
}
```

closeTo is a JUnit/Hamcrest matcher that checks if two doubles are within a certain range. If you were not familiar with that matcher, you could have just subtracted the two sums, taken the absolute value, and compared to a small delta.

JUnit and matchers are briefly covered in an earlier lecture, but you can also pick it up quite quickly from scratch by reading at junit.org.

Comparing Timing of Approaches

```
String message1 = "Sequential sum of %,d numbers is %,.4f.";
String message2 = "Parallel sum of   %,d numbers is %,.4f.";
for (int i=3; i<9; i++) {
  int arraySize = (int)Math.pow(10, i)/2;
  double[] nums = MathUtils.randomNums1(arraySize);
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      double sum = MathUtils.arraySum(nums);
      return(String.format(message1, arraySize, sum));
    }
  });
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      double sum = MathUtils.arraySumParallel1(nums);
      return(String.format(message2, arraySize, sum));
    }
  });
```

This is Java 8 code. In Java 7, the variables message1, message2, arraySize, and nums must be declared final.

Representative Timing Results (Four-Core Machine)

Array Size	Sequential Time (Seconds)	Parallel Time (Seconds)
1,000	0.002	0.001
10,000	0.002	0.002
100,000	0.003	0.003
1,000,000	0.004	0.003
10,000,000	0.011	0.010
100,000,000	0.106	0.055

Conclusion: little benefit of parallel approach except with extremely large arrays.

Times below 0.1 should be treated with suspicion and times below 0.01 should be treated as useless unless examples are run many times and mean and standard deviation are collected. Even with large times, you should run examples repeatedly to see if results are consistent.

Rough Guide to Deciding on Sequential vs. Parallel Approaches

- **Parallel is often better**
  - Problem is large
    - E.g., 5,000,000 element array
  - Computations for smallest size is expensive
    - E.g., sum of expensive operation, finding primes
  - Your computer has many processors
    - Two or more, but more is better
- **Sequential is often better**
  - Problem is small
    - E.g., 5,000 element array
  - Computation for smallest size is fast
    - E.g., sum of doubles
  - Your computer has few processors
    - Obviously, always use sequential for 1-core machines
- **Don't trust your intuition**
  - Time it to be sure

Preview: Summing Arrays in Java 8

Streams

- **Streams are wrappers around Lists**
  - And other data sources
- **Streams have many convenient methods**
  - General Streams (store Objects under the hood)
    - forEach, filter, map, reduce, distinct, limit, etc.
  - DoubleStream, IntStream (store primitives under the hood)
    - Most general methods plus sum, min, max, etc.
- **Streams can run in parallel automatically**
  - anyStream.parallel().anyOperation(…)...;
- **Streams are covered in detail later**
  - Several lectures specifically on Streams

Sequential Sum: Java 7

```
public static double arraySum(double[] nums,
                              int lowerIndex, int upperIndex) {
  double sum = 0;
  for(int i=lowerIndex; i<=upperIndex; i++) {
    sum += nums[i];
  }
  return(sum);
}

public static double arraySum(double[] nums) {
  return(arraySum(nums, 0, nums.length-1));
}
```


Sequential Sum: Java 8

```
public static double arraySum(double[] nums) {
  return(DoubleStream.of(nums).sum());
}
```

Parallel Sum: Java 7 (Part 1)

```
public class ParallelArraySummer extends RecursiveTask<Double> {
  private static final int PARALLEL_CUTOFF = 1000;
  private double[] nums;
  private int lowerIndex, upperIndex;

  public ParallelArraySummer(double[] nums,
                             int lowerIndex, int upperIndex) {
    this.nums = nums;
    this.lowerIndex = lowerIndex;
    this.upperIndex = upperIndex;
  }
```


Parallel Sum: Java 7 (Part 2)

```
  @Override
  protected Double compute() {
    int range = upperIndex - lowerIndex;
    if (range <= PARALLEL_CUTOFF) {
      return(MathUtils.arraySum(nums, lowerIndex, upperIndex));
    } else {
      int middleIndex = lowerIndex + range/2;
      ParallelArraySummer leftSummer =
        new ParallelArraySummer(nums, lowerIndex, middleIndex);
      ParallelArraySummer rightSummer =
        new ParallelArraySummer(nums, middleIndex+1, upperIndex);
      leftSummer.fork();
      Double rightSum = rightSummer.compute();
      Double leftSum = leftSummer.join();
      return(leftSum + rightSum);
    }
  }
}
```

Parallel Sum: Java 7 (Part 3)

```
private static final ForkJoinPool FORK_JOIN_POOL =  new ForkJoinPool();
...

public static Double arraySumParallel(double[] nums) {
  return(FORK_JOIN_POOL.invoke
    (new ParallelArraySummer(nums, 0, nums.length-1)));
}
```


Parallel Sum: Java 8

```
public static double arraySumParallel(double[] nums) {
  return(DoubleStream.of(nums).parallel().sum());
}
```

Parallel version differs from sequential version only by the "parallel()" invocation!
Timing and verification code was identical to that used for the Java 7 version.

Representative Timing Results (Four-Core Machine)

Array Size	Sequential Time (Seconds)	Parallel Time (Seconds)
1,000	0.002	0.001
10,000	0.002	0.002
100,000	0.003	0.002
1,000,000	0.005	0.002
10,000,000	0.045	0.017
100,000,000	0.456	0.158

Conclusion: although parallel streams use fork/join under the hood, the DoubleStream versions are optimized and are usually more efficient than doing fork/join yourself. Not all fork/join problems can be easily represented by Java 8 streams, but if they can be, the Java 8 approach tends to be dramatically simpler and equally or more efficient.

Making Reusable Parallel Array Processor

Big Idea

- **Many problems share characteristics**
  - Take in an array
  - Simple sequential code to compute result for sub-array
  - Break array exactly in half
  - Compute results for each half
  - Combine results and return final answer
- **Simpler parallel code has benefits**
  - Easier and faster to write and debug
    - Easier to test if results are same
    - Easier to test if timing makes parallel approach worthwhile
  - Easier to maintain
    - Overall less complexity, since real parallel code has been tested in reusable superclass

Reusable Parallel Array Processor: Overview

- **Put common operations in the superclass**
  - Break array in half
  - Call fork on left half
  - Call compute on right half
  - Call join on left half
  - Combine results
- **Pass in parts that change**
  - The array
  - The operation to be performed on small arrays
  - The combining operation

ParallelArrayProcessor: Instance Variables

```
public class ParallelArrayProcessor<T,R> extends RecursiveTask<R> {
  public static final ForkJoinPool FORK_JOIN_POOL = new ForkJoinPool();

  private T[] values;
  private int parallelSizeCutoff;
  private SequentialArrayProcessor<T,R> smallSizeProcessor;
  private Combiner<R> valueCombiner;
  private int lowerIndex, upperIndex;
```

This takes an array of T's and returns an R. For example, if you are counting how many random numbers are lower than 0.01, T would be Double (you take in an array of Doubles) and R would be Integer (you return a whole-number count).

The parallelSizeCutoff determines when the compute method will directly compute the result in the current processor (via the SequentialArrayProcessor), and when it will recursively break the problem down and spin off subproblems to other processors.

The SequentialArrayProcessor takes the array and a small range of indices, and calls computeValue to return the answer for the small size.

The Combiner takes the answers for the left and right parts and calls combine to compute the final result.

ParallelArrayProcessor: Constructor

```
 public ParallelArrayProcessor
        (T[] values,
         int parallelSizeCutoff,
         SequentialArrayProcessor<T,R> smallSizeProcessor,
         Combiner<R> valueCombiner,
         int lowerIndex, int upperIndex) {
    this.values = values;
    this.parallelSizeCutoff = parallelSizeCutoff;
    this.smallSizeProcessor = smallSizeProcessor;
    this.valueCombiner = valueCombiner;
    this.lowerIndex = lowerIndex;
    this.upperIndex = upperIndex;
  }
```

The constructor merely stores the input values in instance variables.

ParallelArrayProcessor: Handling Small Size

```
@Override
protected R compute() {
  int range = upperIndex - lowerIndex;
  if (range <= parallelSizeCutoff) {
    return(smallSizeProcessor.computeValue(values,
                                           lowerIndex, upperIndex));
  } else {
    // Large size case shown on next slide
  }
}
```

If the size is small, compute the value directly on the current processor. The way in which to compute this value depends on the situation, so the computation method will be passed in.

ParallelArrayProcessor: Handling Large Size

```
    int middleIndex = lowerIndex + range/2;
    ParallelArrayProcessor<T,R> leftProcessor =
      new ParallelArrayProcessor<>
           (values, parallelSizeCutoff, smallSizeProcessor,
            valueCombiner, lowerIndex, middleIndex);
    ParallelArrayProcessor<T,R> rightProcessor =
      new ParallelArrayProcessor<>
           (values, parallelSizeCutoff, smallSizeProcessor,
            valueCombiner, middleIndex+1, upperIndex);
    leftProcessor.fork();
    R rightValue = rightProcessor.compute();
    R leftValue = leftProcessor.join();
    return(valueCombiner.combine(leftValue, rightValue));
  }
}
```

If the size is large, find the middle of the array, then make subproblems for the left half of the array and the right half of the array. Call fork on the left half to start that subproblem on separate processor. Call compute on the right half to start that subproblem on current processor. Call join on left half to wait for that subproblem to finish and return a value. Finally, use the Combiner to return a final result based on the results of the two subproblems.

SequentialArrayProcessor (For Handling Small Sizes)

```
public interface SequentialArrayProcessor<T,R> {
  R computeValue(T[] values, int lowIndex, int highIndex);
}
```

Like the recursive/parallel version, the simple sequential version takes in an array of T's and returns an R.

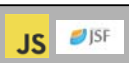
When the size is small, the computeValue method of the specified class that implements this interface is called.

Combiner (For Combining Results from L & R Problems)

```
public interface Combiner<R> {
  R combine(R val1, R val2);
}
```

After both the right (compute) and left (join) values are returned, they are combined via the combine method of the specified class that implements this interface.






Summing Large Array of Doubles: Version 2

Big Idea

- **Repeat the previous problem**
  - But use the ParallelArrayProcessor
  - And Double[] instead of double[]
- **Type of array (T)**
  - Double
- **Type of result (R)**
  - Double (the sum)
- **Sequential process for small size**
  - Loop from one index to the other and sum array values
- **Combiner operation**
  - Add the two values

Core Code

```
public class MathUtils {
  public static Double arraySumParallel(Double[] nums) {
    int parallelSizeCutoff = 1000;
    SequentialArrayProcessor<Double,Double> smallSizeProcessor =
      new ArrayAdder();
    Combiner<Double> valueCombiner = new Adder();
    ForkJoinPool pool = ParallelArrayProcessor.FORK_JOIN_POOL;
    ParallelArrayProcessor<Double,Double> processor =
      new ParallelArrayProcessor<>(nums, parallelSizeCutoff,
                                   smallSizeProcessor, valueCombiner,
                                   0, nums.length-1);
    return(pool.invoke(processor));
  }
```

Cutoff is large (1000), because final computation is very fast.

Computation for small sizes is handled by computeValue method of ArrayAdder, which calls MathUtils.arraySum.

The combination process is handled by the combine method of Adder, which adds the two values.

Handler for Small Sizes

```
public class ArrayAdder implements SequentialArrayProcessor<Double,Double> {
  @Override
  public Double computeValue(Double[] values,
                             int lowIndex, int highIndex) {
    return(MathUtils.arraySum(values, lowIndex, highIndex));
  }
}
```

MathUtils.arraySum was shown in first problem, but is repeated on next slide for reference.

Helper Method: MathUtils.arraySum

```
public static Double arraySum(Double[] nums,
                              int lowerIndex, int upperIndex) {
  double sum = 0;
  for(int i=lowerIndex; i<=upperIndex; i++) {
    sum += nums[i];
  }
  return(sum);
}

public static Double arraySum(Double[] nums) {
  return(arraySum(nums, 0, nums.length-1));
}
```

The version of arraySum that takes only the array (no indices) can be used as a comparison to test that recursive version yields correct answer.

Handler for Combining Values

```
public class Adder implements Combiner<Double> {
  @Override
  public Double combine(Double d1, Double d2) {
    return(d1 + d2);
  }
}
```

Helper Method for Building Arrays (Used by Test Routines)

```
public class MathUtils {
  ...
  public static Double[] randomNums2(int length) {
    Double[] nums = new Double[length];
    for(int i=0; i<length; i++) {
      nums[i] = Math.random();
    }
    return(nums);
  }
}
```


Verifying Both Approaches Give "Same" Answer

```
public class Tester2 {
  ...
  @Test
  public void testSums() {
    for (int i=1; i<5; i++) {
      int arraySize = (int)Math.pow(10, i);
      Double[] nums = MathUtils.randomNums2(arraySize);
      double sum1 = MathUtils.arraySum(nums);
      double sum2 = MathUtils.arraySumParallel(nums);
      assertThat(sum1, is(closeTo(sum2, 0.000001)));
    }
  }
}
```

Same as previous test case, except that it uses arraySumParallel2 (which extends the ParallelArrayProcessor superclass) instead of arraySumParallel1 (which implemented the fork/join approach directly).

Comparing Timing of Approaches

```
String message1 = "Sequential sum of %,d numbers is %,.4f.";
String message2 = "Parallel sum of   %,d numbers is %,.4f.";
for (int i=3; i<9; i++) {
  int arraySize = (int)Math.pow(10, i)/2;
  Double[] nums = MathUtils.randomNums2(arraySize);
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      double sum = MathUtils.arraySum(nums);
      return(String.format(message1, arraySize, sum));
    }
  });
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      double sum = MathUtils.arraySumParallel2(nums);
      return(String.format(message2, arraySize, sum));
    }
  });
```


Same results as previous test case: little advantage for parallel approach on my four-core machine, except for very large array sizes.
This is Java 8 code. In Java 7, the variables message1, message2, arraySize, and nums must be declared final.

Computing Minimum of Expensive Operation

Big Idea

- **Transform each element before taking min**
  - Use an arbitrary function that calls Math.sin, Math.cos, Math.sqrt (etc.) several times
- **Type of array (T)**
  - Double
- **Type of result (R)**
  - Double (the minimum)
- **Sequential process for small size**
  - Loop from one index to the other, apply the function, and keep track of minimum of the results
- **Combiner operation**
  - Returns smaller of the two values

Core Code

```
public static double smallestFancyValueParallel(Double[] nums) {
  int parallelSizeCutoff = 500;
  SequentialArrayProcessor<Double,Double> smallSizeProcessor =
    new FancyArrayMinimizer();
  Combiner<Double> valueCombiner = new Minimizer();
  ForkJoinPool pool = ParallelArrayProcessor.FORK_JOIN_POOL;
  ParallelArrayProcessor<Double,Double> processor =
    new ParallelArrayProcessor<>(nums, parallelSizeCutoff,
                                 smallSizeProcessor, valueCombiner,
                                 0, nums.length-1);
  return(pool.invoke(processor));
}
```

Cutoff is medium-large (500), because final computation is moderately fast.

Computation for small sizes is handled by computeValue method of FancyArrayMinimizer, which calls the transformation function and maintains the minimum of the results.

The combination process is handled by the combine method of Minimizer, which returns the smaller of the two values.

Handler for Small Sizes: Main Class

```
public class FancyArrayMinimizer
       implements SequentialArrayProcessor<Double,Double> {
  @Override
  public Double computeValue(Double[] values, int lowIndex, int highIndex) {
    return(MathUtils.smallestFancyValue(values, lowIndex, highIndex));
  }
}
```


Helper Method: MathUtils.smallestFancyValue

```
  public static double smallestFancyValue(Double[] nums,
                                          int lowerIndex, int upperIndex) {
    Funct<Double,Double> slowFunct = new SlowFunct();
    return(smallestTransformedValue(nums, slowFunct,
                                    lowerIndex, upperIndex));
  }

  public static double smallestFancyValue(Double[] nums) {
    return(smallestFancyValue(nums, 0, nums.length-1));
  }
```

The version of smallestFancyValue that takes only the array (no indices) can be used as a comparison to test that recursive version yields correct answer.

This uses the smallestTransformed value method and the Funct interface, to make the process reusable for other operations. This makes it easy to compare how expensive the transformation operation must be before the parallel version is faster.

Helper Method: MathUtils.smallestTransformedValue

```
public static double smallestTransformedValue
      (Double[] nums,
       Funct<Double,Double> transformer,
       int lowerIndex, int upperIndex) {
  double smallest = Double.MAX_VALUE;
  for (int i=lowerIndex; i<=upperIndex; i++) {
    smallest = Math.min(smallest, transformer.computeValue(nums[i]));
  }
  return(smallest);
}
```

The actual operation applied to each entry in array is given by the computeValue method of the supplied Funct.

Interface for Defining Transformation Operation

```
public interface Funct<T,R> {
  R computeValue(T input);
}
```

In Java 8, you would just use java.util.function.Function, a builtin interface that is mostly equivalent to the above.

Handler for Defining Transformation Operation

```
public class SlowFunct implements Funct<Double,Double> {
  @Override
  public Double computeValue(Double input) {
    return(MathUtils.slowFunction(input));
  };
}
```

```
public class MathUtils {
  ...
  public static double slowFunction(double d) {
    return(Math.sqrt(Math.sqrt(d*Math.PI) +
                     Math.sin(d) + Math.cos(d) +
                     Math.sin(d) + Math.cos(d) +
                     Math.sin(d) + Math.cos(d) +
                     Math.cbrt(d) + Math.sqrt(d) +
                     Math.exp(Math.cbrt(d))));
  }
```


Handler for Combining Values

```
public class Minimizer implements Combiner<Double> {
  @Override
  public Double combine(Double d1, Double d2) {
    return(Math.min(d1, d2));
  }
}
```

Verifying Both Approaches Give Same Answer

```
public class Tester3 {
  ...
  @Test
  public void testSmallest() {
    for (int i=1; i<5; i++) {
      int arraySize = (int)Math.pow(10, i);
      Double[] nums = MathUtils.randomNums2(arraySize);
      Double min1 = MathUtils.smallestFancyValue(nums);
      Double min2 = MathUtils.smallestFancyValueParallel(nums);
      assertThat(min1, is(equalTo(min2)));
    }
  }
```


Comparing Timing of Approaches

```
public static void main(String[] args) {
  String message1 = "Sequential min of %,d numbers is %,.4f.";
  String message2 = "Parallel min of   %,d numbers is %,.4f.";
  for (int i=3; i<8; i++) {
    int arraySize = (int)Math.pow(10, i)/2;
    Double[] nums = MathUtils.randomNums2(arraySize);
    TimingUtils.timeOp(new Op() {
      @Override
      public String runOp() {
        double min = MathUtils.smallestFancyValue(nums);
        return(String.format(message1, arraySize, min));
      }
    });
    TimingUtils.timeOp(new Op() {
      @Override
      public String runOp() {
        double min = MathUtils.smallestFancyValueParallel(nums);
        return(String.format(message2, arraySize, min));
      }
    });
```


This is Java 8 code. In Java 7, the variables message1, message2, arraySize, and nums must be declared final.

Representative Timing Results (Four-Core Machine)

Array Size	Sequential Time (Seconds)	Parallel Time (Seconds)
500	0.007	0.082
5,000	0.008	0.004
50,000	0.023	0.015
500,000	0.185	0.073
5,000,000	1.846	0.700

Conclusion: for large arrays on four-core machine, parallel approach takes about half the time.

On repeated tests, the times for the 500,000 and especially the 5,000,000 entry cases were quite consistent, with the parallel version being twice the speed.

Generating Large Prime Numbers

Big Idea

- **Generate large primes**
  - Pass in empty BigInteger[], fill it in with 100-digit primes
- **Type of array (T)**
  - BigInteger
- **Type of result (R)**
  - Void. The routines fill in array as side effect.
- **Sequential process for small size**
  - Loop from one index to the other, generate prime, store it in location given by the index
- **Combiner operation**
  - Ignores the two values. The routines fill in the array as a side effect.

Core Code

```
public static void findPrimesParallel(BigInteger[] primes) {
  int parallelSizeCutoff = 2;
  SequentialArrayProcessor<BigInteger,Void> smallSizeProcessor =
    new PrimeStorer();
  Combiner<Void> valueCombiner = new Ignorer();
  ForkJoinPool pool = ParallelArrayProcessor.FORK_JOIN_POOL;
  ParallelArrayProcessor<BigInteger,Void> processor =
    new ParallelArrayProcessor<>(primes, parallelSizeCutoff,
                                 smallSizeProcessor, valueCombiner,
                                 0, primes.length-1);
  pool.invoke(processor);
}
```

Cutoff is very small (2), because final computation is very slow.

Computation for small sizes is handled by computeValue method of PrimeStorer, which computes 100-digit primes and stores them in the array at the index specified.

The "combination" process is handled by the combine method of Ignorer, which ignores the values.

Handler for Small Sizes: Main Class

```
public class PrimeStorer
       implements SequentialArrayProcessor<BigInteger,Void> {
  @Override
  public Void computeValue(BigInteger[] primes,
                           int lowerIndex, int upperIndex) {
    MathUtils.findPrimes(primes, lowerIndex, upperIndex);
    return(null);
  }
}
```


Helper Method: MathUtils.findPrimes

```
  public static void findPrimes(BigInteger[] primes,
                                int lowerIndex, int upperIndex) {
    for (int i=lowerIndex; i<=upperIndex; i++) {
      primes[i] = Primes.findPrime(100);
    }
  }

  public static void findPrimes(BigInteger[] primes) {
    findPrimes(primes, 0, primes.length-1);
  }
```

The version of findPrimes that takes only the array (no indices) can be used as a comparison to test that recursive version yields correct answer.

The Primes class is omitted for brevity, but the essence is that findPrime generates a random, odd, 100-digit BigInteger, tests it with isProbablyPrime, and keeps adding 2 until a match is found. The complete source for all examples can be found in the Java tutorial at http://www.coreservlets.com/

Handler for Combining Values

```
public class Ignorer implements Combiner<Void> {
  @Override
  public Void combine(Void v1, Void v2) {
    return(null);
  }
}
```

The handler for small sizes stores the primes into the array, so there is no useful return value.

Verifying Both Approaches Give Same Answer

```
public class Tester4 {
  ...
  @Test
  public void testPrimes() {
    for (int i=1; i<4; i++) {
      int arraySize = (int)Math.pow(10, i);
      BigInteger[] nums = new BigInteger[arraySize];
      MathUtils.findPrimes(nums);
      assertThat(Primes.allPrime(nums), is(true));
      Arrays.fill(nums, null);
      MathUtils.findPrimesParallel(nums);
      assertThat(Primes.allPrime(nums), is(true));
    }
  }
```

Primes.allPrime checks that each entry in array is prime.

Comparing Timing of Approaches (Part 1)

```
public static void main(String[] args) {
  String message1 = "Sequential search for %,d primes.%n" +
                    "First and last are %s and %s.";
  String message2 = "Parallel search for   %,d primes.%n" +
                    "First and last are %s and %s.";
```


Comparing Timing of Approaches (Part 2)

```
for (int i=1; i<5; i++) {
  int arraySize = (int)Math.pow(10, i)/2;
  BigInteger[] primes = new BigInteger[arraySize];
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      MathUtils.findPrimes(primes);
      return(String.format(message1, arraySize, primes[0], primes[arraySize-1]));
    }
  });
  Arrays.fill(primes, null);
  TimingUtils.timeOp(new Op() {
    @Override
    public String runOp() {
      MathUtils.findPrimesParallel(primes);
      return(String.format(message2, arraySize, primes[0], primes[arraySize-1]));
    }
  });
```


This is Java 8 code. In Java 7, the variables message1, message2, arraySize, and nums must be declared final.

Representative Timing Results (Four-Core Machine)

Array Size	Sequential Time (Seconds)	Parallel Time (Seconds)
5	0.483	0.142
50	1.209	0.486
500	11.776	4.058
5,000	124.344	40.246

Conclusion: for virtually all cases, parallel approach takes about one third the time.

Wrap-Up

Summary: Strategies

- **Use parallel approach when**
  - Problem size is large
  - Can be decomposed into smaller pieces
  - Operation at small size is still relatively expensive
  - You have two or (preferably) more cores
- **Test carefully**
  - Verify that both approaches give same (or close enough) answers
    - But remember that reordering multiplication or addition of doubles does not necessarily yield identical results
  - Time both approaches
    - Preferably with real-life problem sizes and system configurations
- **Consider Java 8 parallel streams**
  - As alternative to explicit fork/join, especially when processing arrays or lists

Summary: Pseudo-Code

```
public class MyTask extends RecursiveTask<ResultType> {
  ... // Instance vars that have data and determine size

  public MyTask(...) { ... }  // Set instance vars

  @Override
  protected ResultType compute() {
    if (size < someCutoff) {
      return(directlyCalculatedResult);
    } else {
      MyTask leftTask = new MyTask(...);
      MyTask rightTask = new MyTask(...);
      leftTask.fork();  // Run on separate processor
      ResultType rightResult = rightTask.compute();// Run now
      ResultType leftResult = leftTask.join();   // Wait
      return(someCombinationOf(leftResult, rightResult));
    }
  }
}
```

coreservlets.com – custom onsite training





Questions?

More info:
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training
Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.