Java Fork/Join for Parallel Programming

javacodegeeks.com Fabrizio Chami February 17th, 2011 view original

The last few years a paradigm shift is taking place in the field of computer processors. For years, processor makers consistently delivered increases in clock rates, so developers enjoyed the fact that their single-threaded software executed faster without any effort from their part. Now, processor makers favor multi-core chip designs, and software has to be written in a multi-threaded or multi-process manner to take full advantage of the hardware. Thus, the only way for software developers to catch up is to write applications that leverage parallelism, i.e. engaging multiple CPU cores for handling all tasks rather than a single faster core. Moore's law still applies, but in a different context.
Parallel computing or parallelization is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). In essence, if a CPU intensive problem can be divided in smaller, independent tasks, then those tasks can be assigned to different processors.

Regarding multi-threading and concurrency, Java is very interesting. It has had support for Threads since its beginning and back in the old days you could manipulate thread execution using a low-level approach with the interrupt, join, sleep methods. Moreover, the notify and wait methods that all objects inherit could also be helpful.

It was possible to control application execution in that way, but the process was a bit tedious. And then came the concurrency package in Java 1.5 which provided a higher level framework with which developers could handle threading in a simpler, easier and less error prone way. The package provides a bunch of utility classes commonly useful in concurrent programming.

Over the years, the need for “concurrency-enabled” programs became even bigger, so the platform took this a step further introducing New Concurrency Features for Java SE 7. One of the features is the introduction of the Fork/Join framework, which was intended to be included in the JDK 1.5 version, but finally did not

make it.

The Fork/Join framework is designed to make divide-and-conquer algorithms easy to parallelize. That type of algorithms is perfect for problems that can be divided into two or more sub-problems of the same type. They use recursion to break down the problem to simple tasks until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. An excellent introduction to the Fork/Join approach is the article "Fork-Join Development in Java SE".

As you may have noticed, the Fork/Join approach is similar to MapReduce in that they are both algorithms for parallelizing tasks. One difference however is that Fork/Join tasks will subdivide themselves into smaller tasks only if necessary (if too large), whereas MapReduce algorithms divide up all the work into portions as the first step of their execution.

The Fork/Join Java framework originated as JSR-166 and you can find more information in the Concurrency JSR-166 Interest Site led by Doug Lea. Actually, this is where you have to go if you wish to use the framework and you do not have JDK 7. As mentioned in the site, we can use the related classes in JDK 1.5 and 1.6 without having to install the latest JDK. In order to do so, we have to download the jsr166 jar and launch our JVM using the option -Xbootclasspath/p:jsr166.jar. Note that you may need to precede "jsr166.jar" with its full file path. This has to be done because the JAR contains some classes that override the core Java classes (for example those under the java.util package). Don't forget to bookmark the JSR-166 API docs.

So, let's see how the framework can be used to address a real problem. We are going to create a program that calculates Fibonacci numbers (a classic pedagogical problem) and see how to use the new classes to make it as fast as we can. This problem has also been addressed in the Java Fork/Join + Groovy article.

First we create a class that represents the problem as follows:

```
package com.javacodegeeks.concurrency.forkjoin;

public class FibonacciProblem {
```

```
 public int n;

 public FibonacciProblem(int n) {
  this.n = n;
 }

 public long solve() {
  return fibonacci(n);
 }

 private long fibonacci(int n) {
  System.out.println("Thread: " +
   Thread.currentThread().getName() + " calculates " + n);
  if (n <= 1)
   return n;
  else
   return fibonacci(n-1) + fibonacci(n-2);
 }

}
```

As you can see, we use the recursive version of the solution and this is a typical implementation. (Note that this implementation is highly inefficient since it calculates the same values over and over. In a real-life scenario, the already calculated values should be cached and retrieved in subsequent executions). Let's now see how a single thread approach would look like:

```
package com.javacodegeeks.concurrency.forkjoin;

import org.perf4j.StopWatch;

public class SillyWorker {
```

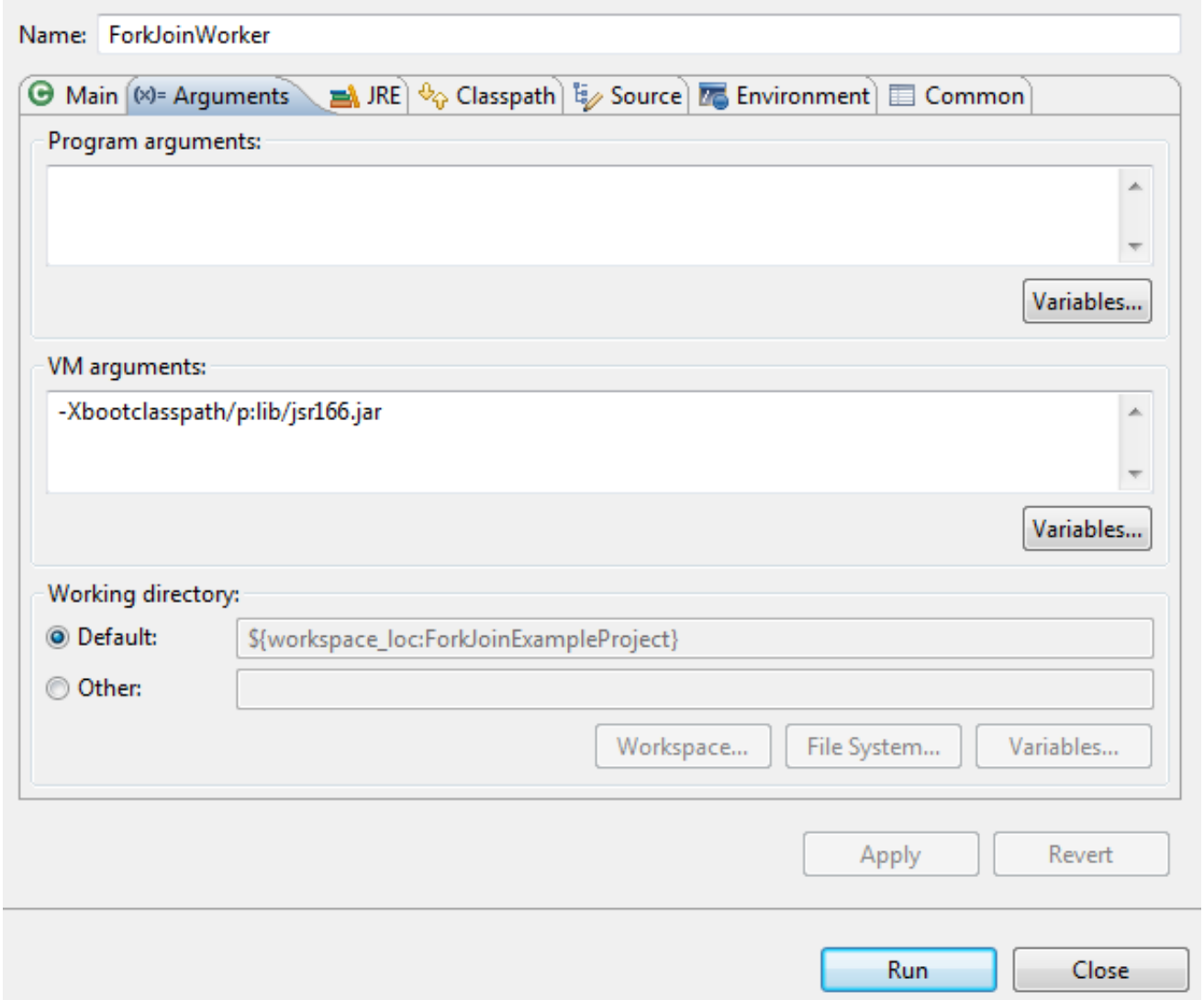
```
 public static void main(String[] args) throws Exception {

  int n = 10;

  StopWatch stopWatch = new StopWatch();
  FibonacciProblem bigProblem = new FibonacciProblem(n);

  long result = bigProblem.solve();
  stopWatch.stop();

  System.out.println("Computing Fib number: " + n);
  System.out.println("Computed Result: " + result);
  System.out.println("Elapsed Time: " +
stopWatch.getElapsedTime());

 }

}
```

We just create a new FibonacciProblem and execute its solve method which will recursively call the fibonacci method. I also use the nice Perf4J library to keep track of the elapsed time. The output would be something like this (I have isolated the last lines): ... Thread: main calculates 1 Thread: main calculates 0 Computing Fib number: 10 Computed Result: 55 Elapsed Time: 8 As expected, all the job is getting done by only one thread (main). Let's see how this can be rewritten using the Fork/Join framework. Note that in the Fibonacci solution, the following take place: fibonacci(n-1) + fibonacci(n-2) Because of that, we can assign each one of these two tasks to a new worker (i.e. a new thread) and after the workers have finished their executions, we will join the result. Taking that into account, we introduce the FibonacciTask class which is essentially a way to divide a big Fibonacci problem to smaller ones.

```
package com.javacodegeeks.concurrency.forkjoin;

import java.util.concurrent.RecursiveTask;
```

```
public class FibonacciTask extends RecursiveTask<Long> {

 private static final long serialVersionUID =
6136927121059165206L;

 private static final int THRESHOLD = 5;

 private FibonacciProblem problem;
 public long result;

 public FibonacciTask(FibonacciProblem problem) {
  this.problem = problem;
 }

 @Override
 public Long compute() {
  if (problem.n < THRESHOLD) { // easy problem, don't
bother with parallelism
   result = problem.solve();
  }
  else {
   FibonacciTask worker1 = new FibonacciTask(new
FibonacciProblem(problem.n-1));
   FibonacciTask worker2 = new FibonacciTask(new
FibonacciProblem(problem.n-2));
   worker1.fork();
   result = worker2.compute() + worker1.join();

  }
  return result;
 }

}
```

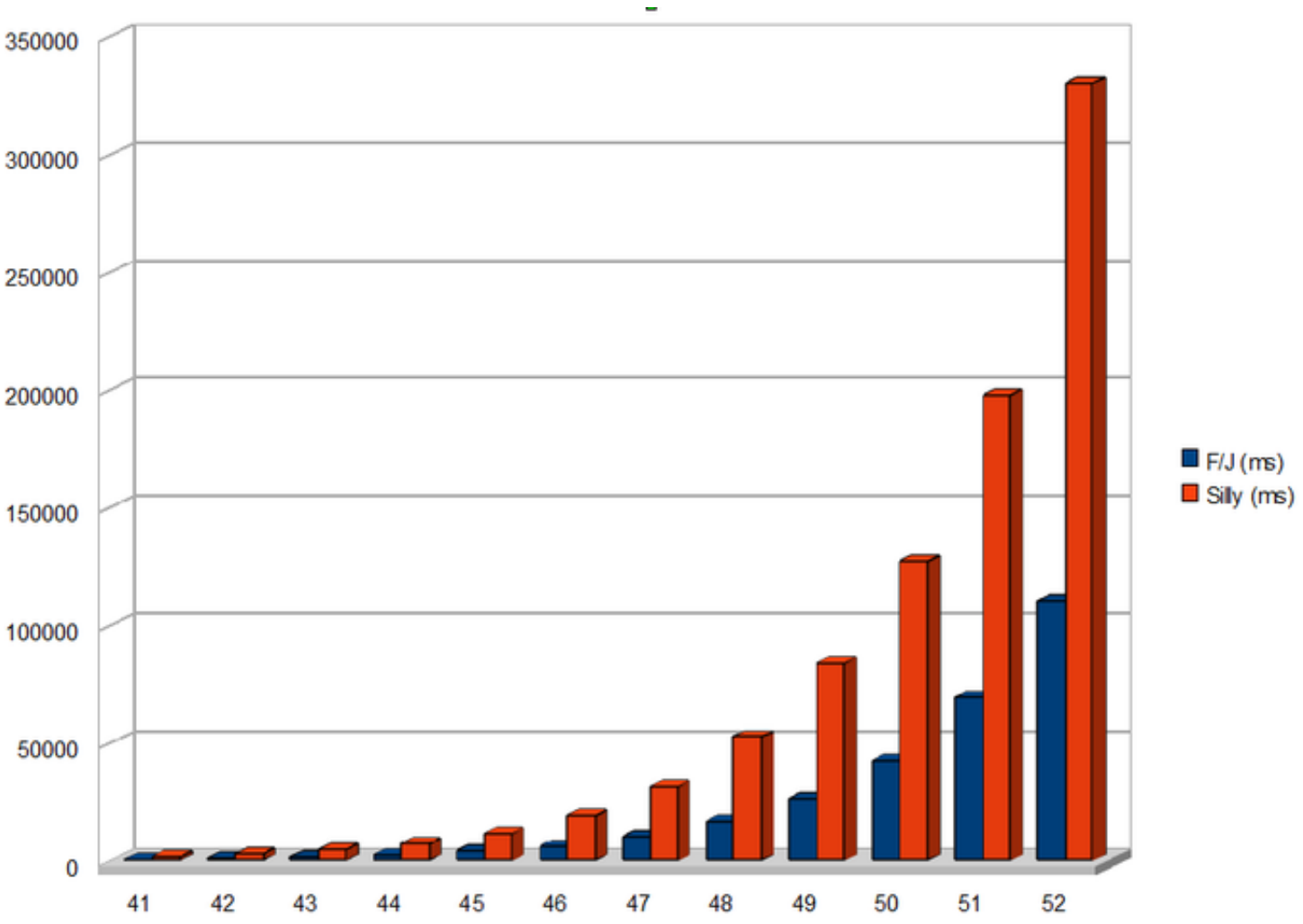
Note: If you are not using JDK 7 and have manually included the JSR-166 libraries, you will have to override the default Java core classes. Otherwise you will encounter the following error: java.lang.SecurityException: Prohibited package name: java.util.concurrent To prevent this, setup your JVM to override the classes by using the following argument: -Xbootclasspath/p:lib/jsr166.jar I have used the “lib/jsr166.jar” value because the JAR resides in a folder named “lib” inside my Eclipse project. Here is how the configuration looks like:



Our task extends the RecursiveTask class which is recursive result-bearing ForkJoinTask. We override the compute method which handles the main computation performed by this task. In that method, we first check if we have to use parallelism (by comparing to a threshold). If this is an easy to perform task, we directly invoke the solve method, otherwise we create two smaller tasks and then execute each one of them independently. The execution occurs into different threads and their results are then combined. This is achieved by using the fork and join methods. Let's test our implementation:

```
package com.javacodegeeks.concurrency.forkjoin;

import java.util.concurrent.ForkJoinPool;

import org.perf4j.StopWatch;

public class ForkJoinWorker {

 public static void main(String[] args) {

  // Check the number of available processors
  int processors =
Runtime.getRuntime().availableProcessors();
  System.out.println("No of processors: " + processors);

  int n = 10;

  StopWatch stopWatch = new StopWatch();
  FibonacciProblem bigProblem = new FibonacciProblem(n);

  FibonacciTask task = new FibonacciTask(bigProblem);
  ForkJoinPool pool = new ForkJoinPool(processors);
  pool.invoke(task);

  long result = task.result;
  System.out.println("Computed Result: " + result);

  stopWatch.stop();
  System.out.println("Elapsed Time: " +
stopWatch.getElapsedTime());

 }

}
```

We first check the available number of processors in the system and then create a new ForkJoinPool with the corresponding level of parallelism. We assign and execute our task by using the invoke method. Here is the output (I have isolated the first and last methods): No of processors: 8 Thread: ForkJoinPool-1-worker-7 calculates 4 Thread: ForkJoinPool-1-worker-6 calculates 4 Thread: ForkJoinPool-1-worker-4 calculates 4 ... Thread: ForkJoinPool-1-worker-2 calculates 1 Thread: ForkJoinPool-1-worker-2 calculates 0 Computed Result: 55 Elapsed Time: 16 Note that now the computation is delegated to a number of worker threads, each one of which is assigned a task smaller than the original one. You may have noticed that the elapsed time is bigger that the previous one. This paradox happens because we have used a low threshold (5) and a low value for n (10). Because of that a big number of threads is created, thus unnecessary delay is introduced. The real power of the framework will become apparent for bigger values of threshold (around 20) and higher values of n (40 and above). I performed some quick stress tests for values n>40 and here is a chart with the results:



It is obvious that the Fork/Join framework scales a lot better that the single-thread approach and executes the

computations in much smaller times. (If you wish to perform some stress tests of your own, don't forget to remove the System.out call in the FibonacciProblem class.) It is also interesting to see some pictures of the CPU usage in my Windows 7 machine (i7-720QM with 4 cores and Hyper-Threading) while each one of the approaches was used.

Single-thread: The total CPU usage remained very low during the execution (never exceed 16%). As you can see, the CPU is under-utilized while the single-thread application struggles to perform all the calculations.

Multi-threaded:

The CPU utilization is much better and all the processors contribute to the total calculation. We have reached the end to my introduction of the Fork/Join framework in Java. Note that I have only scratched the surface here, numerous other features exist and are ready to help us in leveraging the multi-cores CPUs. A new era is emerging, so developers should get familiar with these concepts. As always, you can find here the source code produced for this article. Don't forget to share!