Fork and Join: Java Can Excel at Painless Parallel Programming Too!

By Julien Ponge

How do the new fork/join tasks provided by Java SE 7 make it easier to write parallel programs?

Published July 2011

Downloads:
Java SE 7
Sample Code(Zip)

Multicore processors are now widespread across server, desktop, and laptop hardware. They are also making their way into smaller devices, such as smartphones and tablets. They open new possibilities for concurrent programming because the threads of a process can be executed on several cores in parallel. One important technique for achieving maximal performance in applications is the ability to split intensive tasks into chunks that can be performed in parallel to maximize the use of computational power.

Dealing with concurrent (parallel) programming has traditionally been difficult, because you have to deal with thread synchronization and the pitfalls of shared data. Interest in language-level support for concurrent programming on the Java platform is strong, as proven by the efforts in the Groovy (GPars), Scala, and Clojure communities. These communities all try to provide comprehensive programming models and efficient implementations that mask the pain points associated with multithreaded and distributed applications. The Java language itself should not be considered inferior in this regard. Java Platform, Standard Edition (Java SE) 5 and then Java SE 6 introduced a set of packages providing powerful concurrency building blocks. Java SE 7 further enhanced them by adding support for parallelism

The following article starts with a brief recall of concurrent programming in Java, starting with the low-level mechanisms that have existed since the early releases. It then shows the rich primitives added by the java.util.concurrent packages before presenting fork/join tasks, an essential addition provided in Java SE 7 by the fork/join framework. An example usage of the new APIs is given. Finally, a discussion on the approach precedes the conclusion.

In what follows, we assume that the reader comes from a Java SE 5 or Java SE 6 background. We present a few pragmatic language evolutions of Java SE 7 along the way.

Concurrent Programming in Java

Plain Old Threads

Historically, concurrent programming in Java consisted of writing threads through the `java.lang.Thread` class and the `java.lang.Runnable` interface, then making sure their code behaved in a correct and consistent fashion with respect to shared mutable objects and avoiding incorrect read/write operations while not creating deadlocks induced by race conditions on lock acquisitions. Here is an example of basic thread manipulation:

```
Thread thread = new Thread() {
      @Override public void run() {
         System.out.println(">>> I am running in a separate thread!");
      }
 };
 thread.start();
 thread.join();
```

All the code in this example does is create a thread that prints a string to the standard output stream. The main thread waits for created (child) thread to complete by calling `join()`.

Directly manipulating threads this way is fine for simple examples, but with concurrent programming, such code can quickly become error-prone, especially when several threads need to cooperate to perform a larger task. In such cases, their control flow needs to be coordinated.

For example, the completion of a thread's execution might depend on other threads having completed their execution. The usual well-known example is that of the producer/consumer, because the producer should wait for the consumer if the consumer's queue is full, and the consumer should wait for the producer when empty. This requirement can be addressed through shared state and condition queues, but you still have to use synchronization by using `java.lang.Object.notify()` and `java.lang.Object.wait()` on shared-state objects, which is easy to get wrong.

Finally, a common pitfall is to use synchronize and provide mutual exclusion over large pieces of code or even whole methods. While this approach leads to thread-safe code, it usually yields poor performance due to the limited parallelism that is induced by exclusion being in effect too long.

As is often the case in computing, manipulating low-level primitives to implement complex operations opens the door to mistakes, and as such, developers should seek to encapsulate complexity within efficient, higher-level libraries. Java SE 5 provided us with just that ability.

Rich Primitives with the java.util.concurrent Packages

Java SE 5 introduced a packages family called java.util.concurrent, which was further enhanced by Java SE 6. This packages family offers the following concurrent programming primitives, collections, and features:

Executors, which are an enhancement over plain old threads because they are abstracted from thread pool management. They execute tasks similar to those passed to threads (in fact, instances implementing `java.lang.Runnable` can be wrapped). Several implementations are provided with thread pooling and scheduling strategies. Also, execution results can be fetched both in a synchronous and asynchronous manner.

Thread-safe queues allow for passing data between concurrent tasks. A rich set of implementations is provided with underlying data structures (such as array lists, linked lists, or double-end queues) and concurrent behaviors (such as blocking, supporting priorities, or delays).

Fine-grained specification of time-out delays, because a large portion of the classes found in the `java.util.concurrent` packages exhibit

support for time-out delays. An example is an executor that interrupts tasks execution if the tasks cannot be completed within a bounded timespan.

Rich synchronization patterns that go beyond the mutual exclusion provided by low-level synchronized blocks in Java. These patterns comprise common idioms such as semaphores or synchronization barriers.

Efficient, concurrent data collections (maps, lists, and sets) that often yield superior performance in multithreaded contexts through the use of copy-on-write and fine-grained locks.

Atomic variables that shield developers from the need to perform synchronized access by themselves. These variables wrap common primitive types, such as integers or Booleans, as well as references to other objects.

A wide range of locks that go beyond the lock/notify capabilities offered by intrinsic locks, for example, support for re-entrance, read/write locking, timeouts, or poll-based locking attempts.

As an example, let us consider the following program:

Note: Due to the new integer literals introduced by Java SE 7, underscores can be inserted anywhere to improve readability (for example, 1_000_000).

```
import java.util.*;
import java.util.concurrent.*;
import static java.util.Arrays.asList;

public class Sums {

    static class Sum implements Callable<Long> {
        private final long from;
        private final long to;
            Sum(long from, long to) {
            this.from = from;
            this.to = to;
        }

        @Override
        public Long call() {
            long acc = 0;
            for (long i = from; i <= to; i++) {
                acc = acc + i;
            }
            return acc;
        }
    }

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

        ExecutorService executor = Executors.newFixedThreadPool(2);
        List <Future<Long>> results = executor.invokeAll(asList(
            new Sum(0, 10), new Sum(100, 1_000), new Sum(10_000, 1_000_000)
        ));
        executor.shutdown();

        for (Future<Long> result : results) {
            System.out.println(result.get());
        }
    }
}
```

This example program leverages an executor to compute sums of long integers. The inner Sum class implements the Callable interface that is used by executors for result-bearing computations, and the concurrent work is performed within the `call()` method. The `java.util.concurrent.Executors` class provides several utility methods, such as providing pre-configured executors or wrapping plain old `java.lang.Runnable` objects into instances of Callable. The advantage of using Callable over Runnable is that Callable can explicitly return a value.

This example uses an executor that dispatches work over two threads. The `ExecutorService.invokeAll()` method takes a collection of Callable instances and waits for the completion of all of them before returning. It returns a list of Future objects, which all represent the “future” result of the computation. If we were to work in an asynchronous fashion, we could test each Future object to check whether its corresponding Callable has finished its work and check whether it threw an exception, and we could even cancel it. By contrast, when using plain old threads, you must encode cancellation logic through a shared mutable Boolean and cripple the code with periodic checks over this Boolean. Because `invokeAll()` is blocking, we can directly iterate over the Future instances and fetch their computed sums.

Also note that an executor service must be shut down. If it is not shut down, the Java Virtual Machine will not exit when the main method does, because there will still be active threads around.

Fork/Join Tasks

Overview

Executors are a big step forward compared to plain old threads because executors ease the management of concurrent tasks. Some types of algorithms exist that require tasks to create subtasks and communicate with each other to complete. Those are the “divide and conquer” algorithms, which are also referred to as “map and reduce,” in reference to the eponymous functions in functional languages. The idea is to split the data space to be processed by an algorithm into smaller, independent chunks. That is the “map” phase. In turn, once a set of chunks has been processed, partial results can be collected to form the final result. This is the “reduce” phase.

An easy example would be a huge array of integers for which you would like to compute the sum (see Figure 1). Given that addition is commutative, one may split the array into smaller portions where concurrent threads compute partial sums. The partial sums can then be added to compute the total sum. Because threads can operate independently on different areas of an array for this algorithm, you will see a clear performance boost on multicore architectures compared to a mono-thread algorithm that would iterate over each integer in the array.

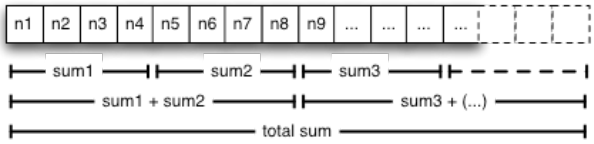

Figure 1: Partial Sums over an Array of Integers

Solving the problem above with executors is easy: Divide the array into the number n of available physical processing units, create Callable instances to compute each partial sum, submit them to an executor managing a pool of n threads, and collect the result to compute the final sum.

On other types of algorithms and data structures, however, the execution plan often is not so simple. In particular, the "map" phase that identifies chunks of data "small enough" to be processed independently in an efficient manner does not know the data space topology in advance. This is especially true for graph-based and tree-based data structures. In those cases, algorithms should create hierarchies of "divisions," waiting for subtasks to complete before returning a partial result. Although less optimal in an array like the one in Figure 1, several levels of concurrent partial-sum computations can be used (for example, divide the array into four subtasks on a dual-core processor).

The problem with the executors for implementing divide and conquer algorithms is not related to creating subtasks, because a `Callable` is free to submit a new subtask to its executor and wait for its result in a synchronous or asynchronous fashion. The issue is that of parallelism: When a `Callable` waits for the result of another `Callable`, it is put in a waiting state, thus wasting an opportunity to handle another `Callable` queued for execution.

The fork/join framework added to the java.util.concurrent package in Java SE 7 through Doug Lea's efforts fills that gap. The Java SE 5 and Java SE 6 versions of `java.util.concurrent` helped in dealing with concurrency, and the additions in Java SE 7 help with parallelism.

Additions for Supporting Parallelism

The core addition is a new `ForkJoinPool` executor that is dedicated to running instances implementing `ForkJoinTask`. `ForkJoinTask` objects support the creation of subtasks plus waiting for the subtasks to complete. With those clear semantics, the executor is able to dispatch tasks among its internal threads pool by "stealing" jobs when a task is waiting for another task to complete and there are pending tasks to be run.

`ForkJoinTask` objects feature two specific methods:

The `fork()` method allows a `ForkJoinTask` to be planned for asynchronous execution. This allows a new `ForkJoinTask` to be launched from an existing one.

In turn, the `join()` method allows a `ForkJoinTask` to wait for the completion of another one.

Cooperation among tasks happens through `fork()` and `join()`, as illustrated in Figure 2. Note that the `fork()` and `join()` method names should not be confused with their POSIX counterparts with which a process can duplicate itself. There, `fork()` only schedules a new task within a `ForkJoinPool`, but no child Java Virtual Machine is ever created.

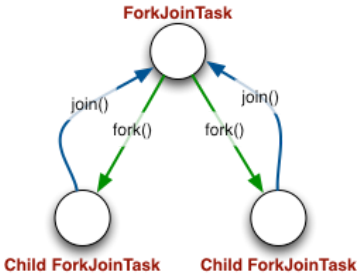

Figure 2: Cooperation Among Fork and Join Tasks

There are two types of `ForkJoinTask` specializations:

Instances of `RecursiveAction` represent executions that do not yield a return value.

In contrast, instances of `RecursiveTask` yield return values.
In general, `RecursiveTask` is preferred because most divide-and-conquer algorithms return a value from a computation over a data set. For the execution of tasks, different synchronous and asynchronous options are provided, making it possible to implement elaborate patterns.

Example: Counting Occurrences of a Word in Documents

To illustrate the usage of the new fork/join framework, let us take a simple example in which we will count the occurrences of a word in a set of documents. First and foremost, fork/join tasks should operate as "pure" in-memory algorithms in which no I/O operations come into play. Also, communication between tasks through shared state should be avoided as much as possible, because that implies that locking might have to be performed. Ideally, tasks communicate only when one task forks another or when one task joins another.

Our application operates on a file directory structure and loads each file's content into memory. Thus, we need the following classes to represent this model. A document is represented as a list of lines:

```
class Document {
    private final List<String> lines;

    Document(List<String> lines) {
        this.lines = lines;
    }

    List<String> getLines() {
        return this.lines;
    }

    static Document fromFile(File file) throws IOException {
        List<String> lines = new LinkedList<>();
        try(BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                lines.add(line);
```

```
                line = reader.readLine();
            }
        }
        return new Document(lines);
    }
}
```

Note: If you are new to Java SE7, you should be surprised by the `fromFile()` method on two accounts:

The `LinkedList` uses the diamond syntax (<>) to let the compiler infer the generic type parameters. Since lines is a `List<String>`, `LinkedList<>` is expanded as `LinkedList<String>`. The diamond operator makes dealing with generics easier by avoiding repeating types when they can easily be inferred at compilation time.

The `try` block uses the new automatic resource management language feature. Any class implementing `java.lang.AutoCloseable` can be used in a try block opening. Regardless of whether an exception is being thrown, any resource declared here will be properly closed when the execution leaves the `try` block. Prior to Java SE 7, properly closing multiple resources quickly turned into a nightmare of nested `if/try/catch/finally` blocks that were often hard to write correctly.

A folder is then a simple tree-based structure:

```
class Folder {
    private final List<Folder> subFolders;
    private final List<Document> documents;

    Folder(List<Folder> subFolders, List<Document> documents) {
        this.subFolders = subFolders;
        this.documents = documents;
    }

    List<Folder> getSubFolders() {
        return this.subFolders;
    }

    List<Document> getDocuments() {
        return this.documents;
    }

    static Folder fromDirectory(File dir) throws IOException {
        List<Document> documents = new LinkedList<>();
        List<Folder> subFolders = new LinkedList<>();
        for (File entry : dir.listFiles()) {
            if (entry.isDirectory()) {
                subFolders.add(Folder.fromDirectory(entry));
            } else {
                documents.add(Document.fromFile(entry));
            }
        }
        return new Folder(subFolders, documents);
    }
}
```

We can now start the implementation of our main class:

```
import java.io.*;
import java.util.*;
import java.util.concurrent.*;

public class WordCounter {

    String[] wordsIn(String line) {
        return line.trim().split("(\\s|\\p{Punct})+");
    }

    Long occurrencesCount(Document document, String searchedWord) {
        long count = 0;
        for (String line : document.getLines()) {
            for (String word : wordsIn(line)) {
                if (searchedWord.equals(word)) {
                    count = count + 1;
                }
            }
        }
        return count;
    }
}
```

The `occurrencesCount` method returns the number of occurrences of a word in a document, leveraging the `wordsIn` method, which yields an array of the words in a line. It does so by splitting the line based on blanks and punctuation characters.

We will implement two types of fork/join tasks. Intuitively, the number of occurrences of a word in a folder is the sum of those in each of its subfolders and documents. Hence, we will have one task for counting the occurrences in a document and one for counting them in a folder. The latter type forks children tasks and then joins them to collect their findings.

The tasks dependency is easy to grasp because it directly maps the underlying document or folder tree structure, as depicted in Figure 3. The fork/join framework maximizes parallelism by ensuring that a pending document's or folder's word counting task can be executed while a folder's task is waiting on a `join()` operation.

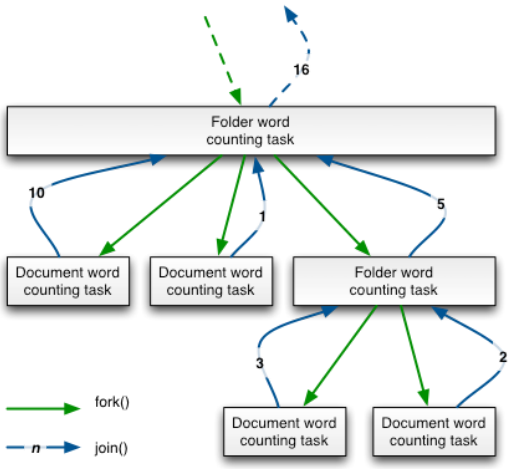

Figure 3: Fork/Join Word Counting Tasks

Let us begin with `DocumentSearchTask`, which counts the occurrences of a word in a document:

```
class DocumentSearchTask extends RecursiveTask<Long> {
    private final Document document;
    private final String searchedWord;

    DocumentSearchTask(Document document, String searchedWord) {
        super();
        this.document = document;
        this.searchedWord = searchedWord;
    }

    @Override
    protected Long compute() {
        return occurrencesCount(document, searchedWord);
    }
}
```

Because our tasks yield values, they extend `RecursiveTask` and take Long as a generic type because the number of occurrences will be represented by
a `long` integer. The `compute()` method is the core of any `RecursiveTask`. Here it simply delegates to the `occurrencesCount()` method above. We
can now tackle the implementation of `FolderSearchTask`, the task that operates on folder elements in our tree structure:

```
class FolderSearchTask extends RecursiveTask<Long> {
    private final Folder folder;
    private final String searchedWord;

    FolderSearchTask(Folder folder, String searchedWord) {
        super();
        this.folder = folder;
        this.searchedWord = searchedWord;
    }

    @Override
    protected Long compute() {
        long count = 0L;
        List<RecursiveTask<Long>> forks = new LinkedList<>();
        for (Folder subFolder : folder.getSubFolders()) {
            FolderSearchTask task = new FolderSearchTask(subFolder, searchedWord);
            forks.add(task);
            task.fork();
        }
        for (Document document : folder.getDocuments()) {
            DocumentSearchTask task = new DocumentSearchTask(document, searchedWord);
            forks.add(task);
            task.fork();
        }
        for (RecursiveTask<Long> task : forks) {
            count = count + task.join();
        }
        return count;
    }
}
```

The implementation of the `compute()` method in this task simply forks document and folder tasks for each element of the folder that it has been passed
through its constructor. It then joins them all to compute its partial sum and returns the partial sum.

We are now missing only a method to bootstrap the word counting operations on the fork/join framework as well as a fork/join pool executor:

```
private final ForkJoinPool forkJoinPool = new ForkJoinPool();

Long countOccurrencesInParallel(Folder folder, String searchedWord) {
    return forkJoinPool.invoke(new FolderSearchTask(folder, searchedWord));
}
```

An initial `FolderSearchTask` bootstraps it all. The `invoke()` method of `ForkJoinPool` allows waiting for the completion of the computation. In the
case above, `ForkJoinPool` is used through its empty constructor. The parallelism will match the number of hardware processing units available (for
example, it will be 2 on machine with a dual-core processor).

We can now write a `main()method` that takes the folder to operate on and the word to search from command-line arguments:

```
public static void main(String[] args) throws IOException {
    WordCounter wordCounter = new WordCounter();
    Folder folder = Folder.fromDirectory(new File(args[0]));
    System.out.println(wordCounter.countOccurrencesOnSingleThread(folder, args[1]));
}
```

The complete source code for this example also includes a more traditional, recursion-based implementation of the same algorithm that works on a single thread:

```
Long countOccurrencesOnSingleThread(Folder folder, String searchedWord) {
    long count = 0;
    for (Folder subFolder : folder.getSubFolders()) {
        count = count + countOccurrencesOnSingleThread(subFolder, searchedWord);
    }
    for (Document document : folder.getDocuments()) {
        count = count + occurrencesCount(document, searchedWord);
    }
    return count;
}
```

Discussion

An informal test was conducted on a Sun Fire T2000 server from Oracle where the number of cores to be available for a Java Virtual Machine could be specified. Both the fork/join and single thread variants of the above example were run to find the number of occurrences of import over the JDK source code files.

The variants ran several times to ensure that the Java Virtual Machine Hotspot optimizations would have enough time to be put into place. The best execution times with 2, 4, 8, and 12 cores were gathered and then the speedup, that is, the ratio (time on a single thread/time on fork-join) was computed.
The results reflected in Figure 4 and Table 1.

As you can see, there is a near-linear speedup in the number of cores with minimal effort, because the fork/join framework takes care of maximizing parallelism.

Table 1: Informal Test Execution Times and Speedup

Number of Cores	Single-Thread Execution Time (ms)	Fork/Join Execution Time (ms)	Speedup
2	18798	11026	1.704879376
4	19473	8329	2.337975747
8	18911	4208	4.494058935
12	19410	2876	6.748956885

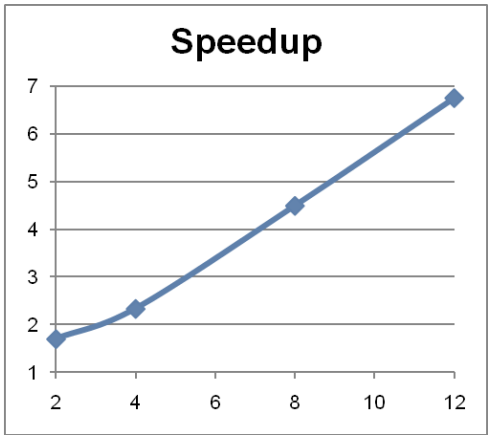

Figure 4: Speedup (Vertical Axis) with Respect to the Number of Cores (Horizontal Axis)

We could have refined the computation to also fork tasks to operate not at the document level, but at the line level. This would have made it possible for concurrent tasks to operate on different lines of the same document. This would, however, be far-fetched. Indeed, a fork/join task should perform a “sufficient” amount of computation to overcome the fork/join thread pool and task management overhead. Working at the line level would be too trivial and hamper the efficiency of the approach.

The included source code also features another fork/join example based on the merge-sort algorithm over arrays of integers. This is interesting because it is implemented using `RecursiveAction`, the fork/join task that does not yield values on join()method invocations. Instead, tasks share mutable state: the array to be sorted. Again, experiments show a near-linear speedup in the number of cores.

Conclusion

This article discussed concurrent programming in Java with a strong focus on the new fork/join tasks provided by Java SE 7 for making it easier to

write parallel programs. The article showed that rich primitives can be used and assembled to write high-performance programs that take advantage of multicore processors, all without having to deal with low-level manipulation of threads and shared state synchronization. The article illustrated the use of those new APIs on a word-occurrence counting example, which is both compelling and easy to grasp. A near-linear speedup was obtained in the number of cores in an informal test. These results show how useful the fork/join framework can be; because we neither had to change the code nor tweak it or the Java Virtual Machine to maximize hardware core utilization.

You can apply this technique to your own problems and data models, too. You should see sensible speedups as long as you can rewrite your algorithms in a “divide and conquer” fashion that is free of I/O work and locking.

Acknowledgements
The author would like to thank Brian Goetz and Mike Duigou for their useful feedback on the early revisions of the article. He would also like to thank Scott Oaks and Alexis Moussine-Pouchkine for their help in running the tests over suitable hardware.

See Also
JavSE Downloads: http://www.oracle.com/technetwork/java/javase/downloads/index.html
Sample Code: http://www.oracle.com/technetwork/articles/java/forkjoinsources-430155.zip
Java SE 7 API: http://download.java.net/jdk7/docs/api/
JSR-166 Interest Site by Doug Lea: http://gee.cs.oswego.edu/dl/concurrency-interest/
Project Coin: http://openjdk.java.net/projects/coin/
Java Concurrency in Practice by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea (Addision-Wesley Professional):
http://www.informit.com/store/product.aspx?isbn=0321349601
Merge-sort algorithm: http://en.wikipedia.org/wiki/Merge_sort
Groovy: http://groovy.codehaus.org/
GPars: http://gpars.codehaus.org/
Scala: http://scala-lang.org
Clojure: http://clojure.org/

Julien Ponge is a long-time open source craftsman. He created the IzPack installer framework and has participated in several other projects, including the GlassFish application server in cooperation with Sun Microsystems. Holding a Ph.D. in computer science from UNSW Sydney and UBP Clermont-Ferrand, he is currently an associate professor in computer science at INSA de Lyon and a researcher as part of the INRIA Amazones team. Speaking both industrial and academic languages, he is highly motivated to further develop synergies between those worlds.