Introduction to the Fork/Join Framework

Esteban Herrera

Nov 4, 2017 • 27 Min read • 5,684 Views

Nov 4, 2017 • 27 Min read • 5,684 Views

Java and J2EE

Introduction

Hardware is getting faster each day.

With multicore processors and GPUs that give us access to parallel programming models, there's been a rise in the popularity of parallel computing platform and APIs like NVIDIA's CUDA.

In Java, the fork/join framework provides support for parallel programming by splitting up a task into smaller tasks to process them using the available CPU cores.

In fact, Java 8's parallel streams and the method Arrays#parallelSort use under the hood the fork/join framework to execute parallel tasks.

Understanding the frame w the framework's inner classes are used, d when it can boost performance.

For reference, you can find the source code of the demo on this GitHub repository.

How does the fork/join framework work?

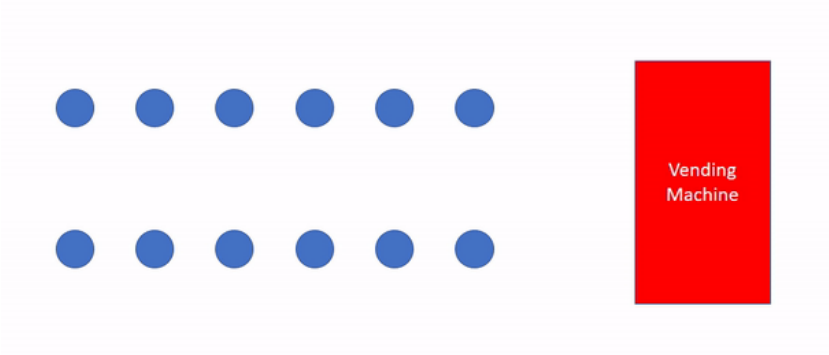
Parallelism is the simultaneous execution of two or more tasks.

However, since parallelism can be considered a concurrency model, it's often confused with the concept of concurrency itself.

Concurrency is the execution of two or more tasks in overlapping time periods that are not necessarily simultaneous.

There's a blog post on the differences between parallelism and concurrency that has great illustrations based on the drawings of Joe Armstrong.

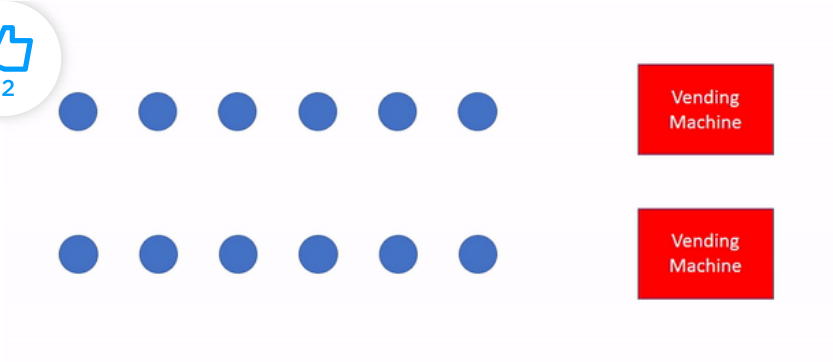
Here's my animated version of those drawings:



Concurrent: 2 queues, 1 vending machine

Understanding the frame

12



Parallel: 2 queues, 2 vending machines

The post goes on noting the differences between these two concepts, especially when talking about event handling systems (like in the vendor-machine example) and computational systems (showing the example of gift-giving).

For our purposes, it's enough to understand that in some problems, concurrency is part of the problem, and in others, concurrency is *not* part of the problem but things like parallelism can be part of the *solution* (by speeding up processing time, for example). However, at the same time, parallelism can bring problems on its own.

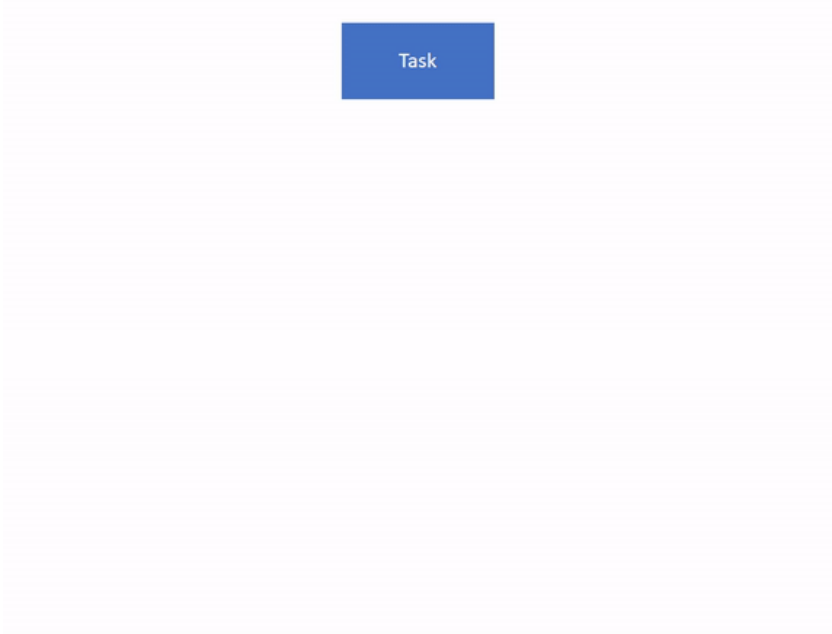
The fork/join framework was designed to speed up the execution of tasks that can be divided into other smaller subtasks, executing them in parallel and then combining their results to get a single one.

For this reason, the subtasks must be independent of each other and the operations must be stateless, making this framework not be the best solution for all problems.

12

aller subtasks until a given threshold is
ached. This is the *fork* part.

Then, the subtasks are processed independently and if they return a result, all the results are recursively combined into a single result. This is the *join* part.



To execution the tasks in parallel, the framework uses a pool of threads, with a number of threads equal to the number of processors available to the Java Virtual Machine (JVM) by default.

Each thread has its own double-ended queue (deque) to store the tasks that will execute.

A deque is a type of queue that supports adding or removing elements from either the front (head) or the back (tail). This allows two things:

- A thread can execute only one task at a time (the task at the head of its deque).

th the work-stealing algorithm, threads that
out of tasks to process can steal tasks from other threads that are still busy (by removing tasks from the tail of their deque).

This approach makes processing more efficient by increasing throughput when there are many tasks to process or when one task diverges into many subtasks.

Understanding the framework classes

The fork/join framework has two main classes, `ForkJoinPool` and `ForkJoinTask`.

`ForkJoinPool` is an implementation of the interface `ExecutorService`. In general, executors provide an easier way to manage concurrent tasks than plain old threads. The main feature of this implementation is the aforementioned work-stealing algorithm.

There's a common `ForkJoinPool` instance available to all applications that you can get with the static method `commonPool()`:

```
1    ForkJoinPool commonPool = ForkJoin
```

The common pool is used by any task that is not explicitly submitted to a specific pool, like the ones used by parallel streams. According to the class documentation, using the common

Understanding the frame non-use, and reinstated upon subsequent e.

12

12

You can also create your own `ForkJoinPool` instance using one of these constructors:

```
1    ForkJoinPool()
2    ForkJoinPool(int parallelism)
3    ForkJoinPool(int parallelism,
4                 ForkJoinPool.ForkJoinW
5                 Thread.UncaughtExceptio
6                 boolean asyncMode
7    )
```

There's another constructor with a lot of more parameters to configure, but most of the time you'll use one of the above.

The first version is the recommended way because it creates an instance with a number of threads equal to the number returned by the method `Runtime.getRuntime().availableProcessors()`, using defaults for all the other parameters.

In the other versions, you can specify the level of parallelism, the factory for creating new threads, the handler for internal worker threads that terminate due to unrecoverable errors that are thrown while executing tasks, and a flag recommended for applications in which worker threads only process event-style asynchronous tasks.

Just like an `ExecutorService` executes an implementation of either the Runnable or the

12

lement by extending one of its two
classes:

- RecursiveAction, which represents tasks that *do not* yield a return value, like a `Runnable`.
- RecursiveTask, which represents tasks that yield return values, like a `Callable`.

These classes contain the `compute()` method, which will be responsible for solving the problem directly or by executing the task in parallel. Most of the time, this method is implemented according to the following pseudo-code:

```
1    if (problem is small)
2      directly solve problem
3    else {
4      split problem into independent pa
5      fork new subtasks to solve each p
6      join all subtasks
7      compose result from subresults
8    }
```

`ForkJoinTask` subclasses also contain the following methods:

- fork(), which allows a `ForkJoinTask` to be scheduled for asynchronous execution (launching a new subtask from an existing one).
- join(), which returns the result of the computation when it is done, allowing a task to wait for the completion of another one.

Understanding the frame

base case. A big task is divided into smaller
ks recursively until the base case is reached.

Each time a task is divided, you call the `fork()` method to place the first subtask in the current thread's deque, and then you call the `compute()` method on the second subtask to recursively process it.

Finally, to get the result of the first subtask you call the `join()` method on this first subtask. This should be the last step because `join()` will block the next program from being processed until until the result is returned.

Thus, the order in which you call the methods is important. If you don't call `fork()` before `join()`, there won't be any result to retrieve. If you call `join()` before `compute()`, the program will perform like if it was executed in one thread and you'll be wasting time.

If you follow the right order, while the second subtask is recursively calculating the value, the first one can be stolen by another thread to process it. This way, when `join()` is finally called, either the result is ready or you don't have to wait a long time to get it.

You can also call the method invokeAll(ForkJoinTask<?>... tasks) to fork and join the task in the right order.

Finally, to submit a task to the thread pool, you can use the execute(ForkJoinTask<?> task) as follows:

12

```
3
4       // Or
5       forkJoinPool.execute(recursiveTask
6       Object result = recursiveTask.join
```

Alternatively, use the submit(ForkJoinTask method (which only differs from the `execute` method in that it returns the submitted task):

```
1       forkJoinPool.execute(recursiveActi
2       // Or if a value is returned
3       Object result = forkJoinPool.execu
```

However, you will typically use invoke(ForkJoinTask, which performs the given task, returning its result upon completion:

```
1       forkJoinPool.invoke(recursiveActio
2       // Or if a value is returned
3       Object result = forkJoinPool.invok
```

Now let's review an example to get a better grasp on this framework.

Implementing a demo

Let's implement something simple, like finding the sum of all the elements in a list.

This list can be split up in many sublists to sum the elements of each one. Then, we can find sum all those values.

Understanding the frame

12

st, let's create a class that extends from `cursiveAction`:

```
public class SumAction extends Recu

}
```

Next, let's choose a value that will indicate if the task is processed sequentially or in parallel.

The most basic case is when we have a list of two values. However, having subtasks that are so small can have a negative impact on performance because forking too much creates significant overhead cost through the recursive stack.

For this reason, we have to choose a value that represents the number of elements that can be processed sequentially without any problem. A value neither too small nor too big.

For this simple example, let's say a list of five elements is the right threshold:

```
public class SumAction extends Recu
  private static final int SEQUENT
}
```

Since the `compute()` method doesn't take parameters, you have to pass to the class constructor the data to work and save it as an instance variable:


```
4       private List<Long> data;
5
6       public SumAction(List<Long> data
7         this.data = data;
8       }
9     }
```

For each recursive call, we can create sublist without having to create a new list every time (remember that the `sublist` method returns a *subview* of the original list and not a copy). If we were working with arrays, probably it would be better to pass the whole array and the start and end index instead of creating smaller copies of the original array each time.

So the `compute()` method looks like this:

```
1     public class SumAction extends Re
2       // ...
3
4       @Override
5       protected void compute() {
6         if (data.size() <= SEQUENTIAL_
7           long sum = computeSumDirect
8           System.out.format("Sum of %
9         } else { // recursive case
10          // Calcuate new range
11          int mid = data.size() / 2;
12          SumAction firstSubtask =
13              new SumAction(data.subL
14          SumAction secondSubtask =
15              new SumAction(data.subL
16
17          firstSubtask.fork(); // que
18          secondSubtask.compute(); //
19          firstSubtask.join(); // wai
```

12

```
23          }
24        }
25      }
```

If the size of the list is equal or less than the threshold, the sum is computed directly and the result is printed.

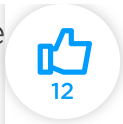
Otherwise, the list is divided into two `SumAction` tasks. Then, the first task is forked while the result of the second is computed (this is the recursive call until the condition of the base case is fulfilled) and after that, we wait for the first task result.

The method to compute the sum can be as simple as:

```
1     public class SumAction extends Re
2       // ...
3
4       private long computeSumDirectly
5         long sum = 0;
6         for (Long l: data) {
7           sum += l;
8         }
9         return sum;
10      }
11    }
```

Finally, let's add a main method to execute the class:

```
1     public class SumAction extends Re
2       // ...
3
```




```
7         List<Long> data = random
8             .longs(10, 1, 5)
9             .boxed()
10            .collect(toList());
11
12        ForkJoinPool pool = new ForkJ
13        SumAction task = new SumActio
14        pool.invoke(task);
15      }
16    }
```

In this method, a list of ten random numbers from one to four (the third parameter of the method `longs` represents the exclusive bound of the range) is generated and passed to a `SumAction` instance, which in turn is passed to a new `ForkJoinPool` instance to be executed.

If we run the program, this could be a possible output:

```
Sum of [1, 4, 4, 2, 3]: 14
Sum of [4, 1, 2, 1, 1]: 9
```

However, dividing the task may not always result in evenly distributed subtasks. For example, if we try with a list of eleven elements, this could be a possible output:

```
Sum of [1, 2, 2]: 5
Sum of [3, 3, 2]: 8
Sum of [2, 4, 1, 3, 3]: 13
```

Now, let's create a version of this class that extends from `RecursiveTask` and returns the

12

```
lic class SumTask extends RecursiveTask
1
2     }
3
```

We can copy the instance variables, the constructor and the `computeSumDirectly()` method:

```
1     public class SumTask extends Recu
2       private static final int SEQUEN
3
4       private List<Long> data;
5
6       public SumTask(List<Long> data)
7         this.data = data;
8       }
9
10      // ...
11
12      private long computeSumDirectly
13        long sum = 0;
14        for (Long l: data) {
15          sum += l;
16        }
17        return sum;
18      }
19    }
```

Changing the `compute()` method a little bit to return the sum value:

```
1     public class SumTask extends Recu
2       // ...
3
```

Understanding the frame

12

12


```
7           long sum = computeSumDirect
8           System.out.format("Sum of %
9           return sum;
10        } else { // recursive case
11          // Calcuate new range
12          int mid = data.size() / 2;
13          SumTask firstSubtask =
14              new SumTask(data.subLis
15          SumTask secondSubtask =
16              new SumTask(data.subLis
17
18          // queue the first task
19          firstSubtask.fork();
20
21          // Return the sum of all sul
22          return secondSubtask.compute
23                  +
24                  firstSubtask.join()
25        }
26      }
27
28      // ...
29    }
```

In its `main()` method, we only need to print the returned value from the task:

```
1     public class SumTask extends Recu
2       // ...
3
4       public static void main(String[
5         Random random = new Random();
6
7         List<Long> data = random
8             .longs(10, 1, 5)
9             .boxed()
10            .collect(toList());
11
```

```
15        }
16    }
```

When we run the program, the following could be a possible output:

```
Sum of [4, 3, 1, 1, 1]: 10
Sum of [1, 1, 1, 2, 1]: 6
Sum: 16
```

Testing performance

I'm going to run an informal test to see the performance of the class `SumTask` versus a sequential implementation, on a list of ten million elements:

```
public class ComparePerformance {

  public static void main(String[
    Random random = new Random();

    List<Long> data = random
        .longs(10_000_000, 1, 100
        .boxed()
        .collect(toList());

    testForkJoin(data);
    //testSequentially(data);
  }

  private static void testForkJoi
    final long start = System.cur

```

12

```
21
22          System.out.println("Executed
23        }
24
25        private static void testSequent
26          final long start = System.cur
27
28          long sum = 0;
29          for (Long l: data) {
30            sum += l;
31          }
32
33          System.out.println("Executed
34        }
35      }
```

I know that the validity of a test like this is questionable (using `System.currentTimeMillis()` to measure the execution time?), and that depending on the hardware you're testing, you can get different results.

However, we need not worry too much about the actual numbers because how they compare to each other is far more interesting!

I ran the program ten times for each test to get an average execution time. the first test used the fork/join framework, using a threshold of one thousand elements, then a one hundred thousand threshold, and then a one million elements threshold. The second test used the sequential implementation.

Here are the results:

12

	ForkJoin 1,000 TH	ForkJoin 100,000 TH	ForkJoin 1,000,000 TH	Seq
Time #01 (in ms)	36	26	34	15
Time #02 (in ms)	107	31	36	16
Time #03 (in ms)	38	25	28	16
Time #04 (in ms)	34	31	33	31
Time #05 (in ms)	32	31	29	15
Time #06 (in ms)	140	27	30	16
Time #07 (in ms)	35	26	30	32
Time #08 (in ms)	36	28	36	16
Time #09 (in ms)	34	27	29	16

12

	ForkJoin 1,000 TH	ForkJoin 100,000 TH	ForkJoin 1,000,000 TH	Seq
Time #10 (in ms)	37	32	33	15
Average	**52.9**	**28.4**	**31.8**	**18.8**

This gave some unexpected results.

First, you can see that on a list of ten million, one thousand elements is actually a low threshold, which impacts performance due to the overhead of creating many small subtasks.

On the other hand, if you go all the way up to one million, you'll get better times in average, but the sweet spot is more around one hundred thousand.

However, the most time efficient implementation was the sequential one, about 30% faster than the fork/join implementation with a one hundred thousand threshold.

Is it because the task is so simple it need not be executed in parallel?

Probably. Although a sum task can be considered a divide and conquer algorithm (the type of algorithm that works perfectly with the fork/join framework), if the task is small, then it is probably better to do it sequentially; at the end, the cost of splitting and queuing the tasks adds up to exacerbate the runtime.

12

same loop than the sequential implementation to sum all the list elements, but what happens if we change the sequential implementation to something like the following:

```
1    private static void testSequential
2      final long start = System.curren
3
4      data.stream().reduce(0L, Long::su
5
6      System.out.println("Executed wit
7    }
```

The code looks more elegant, but in my case, the average execution went up to 241.5 ms, probably due to the overhead of creating the stream.

And when I use a parallel stream (which use the fork/join framework under the hood with the common pool):

```
1    private static void testParallelSt
2      final long start = System.curren
3
4      data.parallelStream().reduce(0L,
5
6      System.out.println("Executed wit
7    }
```

The average execution time I get is 181.6 ms.

So the implementation also matters.

But back to the original comparison, is it because my processor only has four cores and

re cores — maybe sixteen of them — could rk best for parallelism as smaller tasks can be pipelined quickly.

However, what we can deny is that two very important aspects of getting a good performance when using the fork/join framework are:

- Determine the proper threshold of a fork-join task
- Determine the right level of parallelism

In other words, deciding whether to use fork/join comes down to experimenting with different parameters.

David Hovemeyer says in his lecture about Fork/join parallelism:

> *One of the main things to consider when implementing an algorithm using fork/join parallelism is choosing the threshold which determines whether a task will execute a sequential computation rather than forking parallel sub-tasks.*
>
> *If the threshold is too large, then the program might not create enough tasks to fully take advantage of the available processors/cores.*
>
> *If the threshold is too small, then the overhead of task creation and management could become significant.*
>
> *In general, some experimentation will be*

arly parallelism has advantages and
advantages, and it's not a good fit for all
situations.

Conclusion

You have learned about the fork/join framework, which is designed to work with large tasks that can be recursively split up into smaller tasks (the fork part) for processing and the results can be combined at the end (the join part).

The main class of the fork/join framework is `ForkJoinPool`, a subclass of `ExecutorService` that implements a work-stealing algorithm. This algorithm enables a free thread to *steal* the pending work of busier threads.

But the most important thing to really take advantage of this framework is to determine the proper threshold to execute a task in sequence or in parallel and get the right level of parallelism.

Unfortunately, there's no concrete formula for identifying this value. While we may be able to dig deeper and calculate an asymptotic bound on the runtime for parallel processing, the most practical way to figure out when to use fork/join is to experiment. Depending on the task and the hardware used, there will be a variety of results which will provide insight on whether parallelizing is the way to go.

Understanding the frame

12

- Introduction
- How does the fork/jo
- Understanding the fr
- Implementing a dem
- Testing performance
- Conclusion
- Top ^

12

demonstrates the concepts behind the mework; you can find more professional benchmarks, like the one in the post Fork/Join Framework vs. Parallel Streams vs. ExecutorService: The Ultimate Fork/Join Benchmark if this guide has piqued your interest.

Remember that you can find the code of the demo on this GitHub repository. Additional notes on multithreading and multi-core processing are available on this presentation by UNC's CS department.

Thanks for reading!

12