Extracted from:

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

This PDF file contains pages extracted from *Programming Concurrency on the JVM*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com .

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

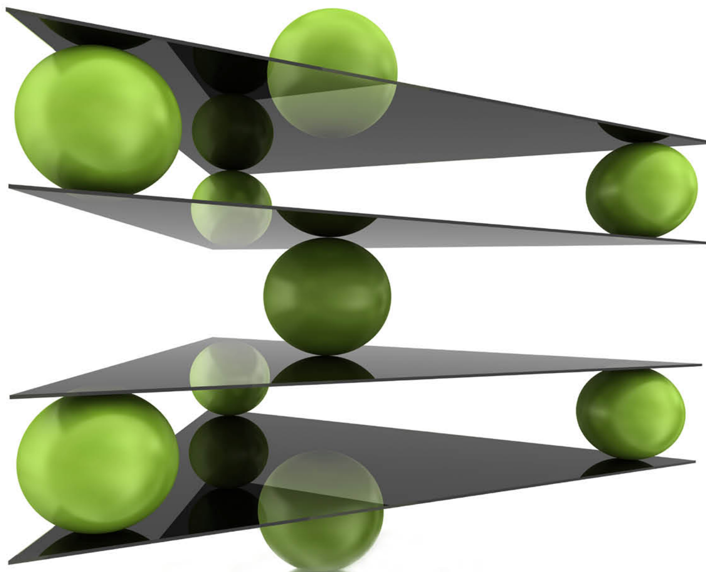
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors



Venkat Subramaniam

edited by Brian P. Hogan

Programming Concurrency on the JVM

Mastering Synchronization, STM, and Actors

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2011 Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-934356-76-0
Printed on acid-free paper.
Book version: P1.0—August 2011

To Mom and Dad, for teaching the values of integrity, honesty, and diligence.

The long-awaited multicore processor arrives tomorrow, and you can't wait to see how the app you're building runs on it. You've run it several times on a single core, but you're eager to see the speedup on the new machine. Is the speedup going to be in proportion to the number of cores? More? Less? A lot less? I've been there and have felt the struggle to arrive at a reasonable expectation.

You should've seen my face the first time I ran my code on a multicore and it performed much worse than I had expected. How could more cores yield slower speed? That was years ago, and I've grown wiser since and learned a few lessons along the way. Now I have better instinct and ways to gauge speedup that I'd like to share with you in this chapter.

2.1 From Sequential to Concurrent

We can't run a single-threaded application on a multicore processor and expect better results. We have to divide it and run multiple tasks concurrently. But, programs don't divide the same way and benefit from the same number of threads.

I have worked on scientific applications that are computation intensive and also on business applications that are IO intensive because they involve file, database, and web service calls. The nature of these two types of applications is different and so are the ways to make them concurrent.

We'll work with two types of applications in this chapter. The first one is an IO-intensive application that will compute the net asset value for a wealthy user. The second one will compute the total number of primes within a range of numbers—a rather simple but quite useful example of a concurrent computation–intensive program. These two applications will help us learn how many threads to create, how to divide the problem, and how much speedup to expect.

Divide and Conquer

If we have hundreds of stocks to process, fetching them one at a time would be the easiest way…to lose the job. The user would stand fuming while our application chugs away processing each stock sequentially.

To speed up our programs, we need to divide the problem into concurrently running tasks. That involves creating these parts or tasks and delegating them to threads so they can run concurrently. For a large problem, we may create as many parts as we like, but we can't create too many threads because we have limited resources.

Determining the Number of Threads

For a large problem, we'd want to have at least as many threads as the number of available cores. This will ensure that as many cores as available to the process are put to work to solve our problem. We can easily find the number of available cores; all we need is a simple call from the code:[1]

```
Runtime.getRuntime().availableProcessors();
```

So, the minimum number of threads is equal to the number of available cores. If all tasks are computation intensive, then this is all we need. Having more threads will actually hurt in this case because cores would be context switching between threads when there is still work to do. If tasks are IO intensive, then we should have more threads.

When a task performs an IO operation, its thread gets blocked. The processor immediately context switches to run other eligible threads. If we had only as many threads as the number of available cores, even though we have tasks to perform, they can't run because we haven't scheduled them on threads for the processors to pick up.

If tasks spend 50 percent of the time being blocked, then the number of threads should be twice the number of available cores. If they spend less time being blocked—that is, they're computation intensive—then we should have fewer threads but no less than the number of cores. If they spend more time being blocked—that is, they're IO intensive—then we should have more threads, specifically, several multiples of the number of cores.

So, we can compute the total number of threads we'd need as follows:

```
Number of threads = Number of Available Cores / (1 - Blocking Coefficient)
```

where the blocking coefficient is between 0 and 1.

A computation-intensive task has a blocking coefficient of 0, whereas an IO-intensive task has a value close to 1—a fully blocked task is doomed, so we don't have to worry about the value reaching 1.

To determine the number of threads, we need to know two things:

- The number of available cores
- The blocking coefficient of tasks

The first one is easy to determine; we can look up that information, even at runtime, as we saw earlier. It takes a bit of effort to determine the blocking

---

1. availableProcessors() reports the number of logical processors available to the JVM.

coefficient. We can try to guess it, or we can use profiling tools or the `java.lang.management` API to determine the amount of time a thread spends on system/IO operations vs. on CPU-intensive tasks.

Determining the Number of Parts

We know how to compute the number of threads for concurrent applications. Now we have to decide how to divide the problem. Each part will be run concurrently, so, on first thought, we could have as many parts as the number of threads. That's a good start but not adequate; we've ignored the nature of the problem being solved.

In the net asset value application, the effort to fetch the price for each stock is the same. So, dividing the total number of stocks into as many groups as the number of threads should be enough.

However, in the primes application, the effort to determine whether a number is prime is not the same for all numbers. Even numbers fizzle out rather quickly, and larger primes take more time than smaller primes. Taking the range of numbers and slicing them into as many groups as the number of threads would not help us get good performance. Some tasks would finish faster than others and poorly utilize the cores.

In other words, we'd want the parts to have even work distribution. We could spend a lot of time and effort to divide the problem so the parts have a fair distribution of load. However, there would be two problems. First, this would be hard; it would take a lot of effort and time. Second, the code to divide the problem into equal parts and distribute it across the threads would be complex.

It turns out that keeping the cores busy on the problem is more beneficial than even distribution of load across parts. When there's work left to be done, we need to ensure no available core is left to idle, from the process point of view. So, rather than splitting hairs over an even distribution of load across parts, we can achieve this by creating far more parts than the number of threads. Set the number of parts large enough so there's enough work for all the available cores to perform on the program.

2.2 Concurrency in IO-Intensive Apps

An IO-intensive application has a large blocking coefficient and will benefit from more threads than the number of available cores.

Let's build the financial application I mentioned earlier. The (rich) users of the application want to determine the total net asset value of their shares

at any given time. Let's work with one user who has shares in forty stocks. We are given the ticker symbols and the number of shares for each stock. From the Web, we need to fetch the price of each share for each symbol. Let's take a look at writing the code for calculating the net asset value.

Sequential Computation of Net Asset Value

As the first order of business, we need the price for ticker symbols. Thankfully, Yahoo provides historic data we need. Here is the code to communicate with Yahoo's financial web service to get the last trading price for a ticker symbol (as of the previous day):

Download divideAndConquer/YahooFinance.java

```
public class YahooFinance {
  public static double getPrice(final String ticker) throws IOException {
    final URL url =
      new URL("http://ichart.finance.yahoo.com/table.csv?s=" + ticker);

    final BufferedReader reader = new BufferedReader(
      new InputStreamReader(url.openStream()));

    //Date,Open,High,Low,Close,Volume,Adj Close
    //2011-03-17,336.83,339.61,330.66,334.64,23519400,334.64
    final String discardHeader = reader.readLine();
    final String data = reader.readLine();
    final String[] dataItems = data.split(",");
    final double priceIsTheLastValue =
      Double.valueOf(dataItems[dataItems.length - 1]);
    return priceIsTheLastValue;
  }
}
```

We send a request to http://ichart.finance.yahoo.com and parse the result to obtain the price.

Next, we get the price for each of the stocks our user owns and display the total net asset value. In addition, we display the time it took for completing this operation.

Download divideAndConquer/AbstractNAV.java

```
public abstract class AbstractNAV {
  public static Map<String, Integer> readTickers() throws IOException {
    final BufferedReader reader =
      new BufferedReader(new FileReader("stocks.txt"));

    final Map<String, Integer> stocks = new HashMap<String, Integer>();

    String stockInfo = null;
    while((stockInfo = reader.readLine()) != null) {
```

• CLICK HERE to purchase this book now. discuss

```
      final String[] stockInfoData = stockInfo.split(",");
      final String stockTicker = stockInfoData[0];
      final Integer quantity = Integer.valueOf(stockInfoData[1]);

      stocks.put(stockTicker, quantity);
    }

    return stocks;
  }

  public void timeAndComputeValue()
    throws ExecutionException, InterruptedException, IOException {
    final long start = System.nanoTime();

    final Map<String, Integer> stocks = readTickers();
    final double nav = computeNetAssetValue(stocks);

    final long end = System.nanoTime();

    final String value = new DecimalFormat("$##,##0.00").format(nav);
    System.out.println("Your net asset value is " + value);
    System.out.println("Time (seconds) taken " + (end - start)/1.0e9);
  }

  public abstract double computeNetAssetValue(
    final Map<String, Integer> stocks)
    throws ExecutionException, InterruptedException, IOException;
}
```

The readTickers() method of AbstractNAV reads the ticker symbol and the number of shares owned for each symbol from a file called stocks.txt, part of which is shown next:

```
AAPL,2505
AMGN,3406
AMZN,9354
BAC,9839
BMY,5099
...
```

The timeAndComputeValue() times the call to the abstract method computeNetAssetValue(), which will be implemented in a derived class. Then, it prints the total net asset value and the time it took to compute that.

Finally, we need to contact Yahoo Finance and compute the total net asset value. Let's do that sequentially:

Download divideAndConquer/SequentialNAV.java

```
public class SequentialNAV extends AbstractNAV {
  public double computeNetAssetValue(
```

```
    final Map<String, Integer> stocks) throws IOException {
    double netAssetValue = 0.0;
    for(String ticker : stocks.keySet()) {
      netAssetValue += stocks.get(ticker) * YahooFinance.getPrice(ticker);
    }
    return netAssetValue;
  }

  public static void main(final String[] args)
    throws ExecutionException, IOException, InterruptedException {
    new SequentialNAV().timeAndComputeValue();
  }
}
```

Let's run the SequentialNAV code and observe the output:

```
Your net asset value is $13,661,010.17
Time (seconds) taken 19.776223
```

The good news is we managed to help our user with the total asset value. However, our user is not very pleased. The displeasure may be partly because of the market conditions, but really it's mostly because of the wait incurred; it took close to twenty seconds[2] on my computer, with the network delay at the time of run, to get the results for only forty stocks. I'm sure making this application concurrent will help with speedup and having a happier user.

Determining Number of Threads and Parts for Net Asset Value

The application has very little computation to perform and spends most of the time waiting for responses from the Web. There is really no reason to wait for one response to arrive before sending the next request. So, this application is a good candidate for concurrency: we'll likely get a good bump in speed.

In the sample run, we had forty stocks, but in reality we may have a higher number of stocks, even hundreds. We must first decide on the number of divisions and the number of threads to use. Web services (in this case, Yahoo Finance) are quite capable of receiving and processing concurrent requests.[3] So, our client side sets the real limit on the number of threads. Since the web service requests will spend a lot of time waiting on a response, the blocking coefficient is fairly high, and therefore we can bump up the number of threads by several factors of the number of cores. Let's say the blocking

2. More than a couple of seconds of delay feels like eternity to users.
3. To prevent denial-of-service attacks (and to up-sell premium services), web services may restrict the number of concurrent requests from the same client. You may notice this with Yahoo Finance when you exceed fifty concurrent requests.

coefficient is 0.9—each task blocks 90 percent of the time and works only 10 percent of its lifetime. Then on two cores, we can have (using the formula from *Determining the Number of Threads*, on page 6) twenty threads. On an eight-core processor, we can go up to eighty threads, assuming we have a lot of ticker symbols.

As far as the number of divisions, the workload is basically the same for each stock. So, we can simply have as many parts as we have stocks and schedule them over the number of threads.

Let's make the application concurrent and then study the effect of threads and partitions on the code.

Concurrent Computation of Net Asset Value

There are two challenges now. First, we have to schedule the parts across threads. Second, we have to receive the partial results from each part to calculate the total asset value.

We may have as many divisions as the number of stocks for this problem. We need to maintain a pool of threads to schedule these divisions on. Rather than creating and managing individual threads, it's better to use a thread pool—they have better life cycle and resource management, reduce startup and teardown costs, and are warm and ready to quickly start scheduled tasks.

As Java programmers, we're used to Thread and synchronized, but we have some alternatives to these since the arrival of Java 5—see *Is There a Reason to Use the Old Threading API in Java?*, on page 12. The new-generation Java concurrency API in java.util.concurrent is far superior.

In the modern concurrency API, the Executors class serves as a factory to create different types of thread pools that we can manage using the ExecutorService interface. Some of the flavors include a single-threaded pool that runs all scheduled tasks in a single thread, one after another. A fixed threaded pool allows us to configure the pool size and concurrently runs, in one of the available threads, the tasks we throw at it. If there are more tasks than threads, the tasks are queued for execution, and each queued task is run as soon as a thread is available. A cached threaded pool will create threads as needed and will reuse existing threads if possible. If no activity is scheduled on a thread for well over a minute, it will start shutting down the inactive threads.

The fixed threaded pool fits the bill well for the pool of threads we need in the net asset value application. Based on the number of cores and the pre-

Joe asks:

Is There a Reason to Use the Old Threading API in Java?

The old threading API has several deficiencies. We'd use and throw away the instances of the Thread class since they don't allow restart. To handle multiple tasks, we typically create multiple threads instead of reusing them. If we decide to schedule multiple tasks on a thread, we had to write quite a bit of extra code to manage that. Either way was not efficient and scalable.

Methods like wait() and notify() require synchronization and are quite hard to get right when used to communicate between threads. The join() method leads us to be concerned about the death of a thread rather than a task being accomplished.

In addition, the synchronized keyword lacks granularity. It doesn't give us a way to time out if we do not gain the lock. It also doesn't allow concurrent multiple readers. Furthermore, it is very difficult to unit test for thread safety if we use synchronized.

The newer generation of concurrency APIs in the java.util.concurrent package, spearheaded by Doug Lea, among others, has nicely replaced the old threading API.

- Wherever we use the Thread class and its methods, we can now rely upon the ExecutorService class and related classes.
- If we need better control over acquiring locks, we can rely upon the Lock interface and its methods.
- Wherever we use wait/notify, we can now use synchronizers such as CyclicBarrier and CountdownLatch.

sumed blocking coefficient, we decide the thread pool size. The threads in this pool will execute the tasks that belong to each part. In the sample run, we had forty stocks; if we create twenty threads (for a two-core processor), then half the parts get scheduled right away. The other half are enqueued and run as soon as threads become available. This will take little effort on our part; let's write the code to get this stock price concurrently.

Download divideAndConquer/ConcurrentNAV.java

```
Line 1 public class ConcurrentNAV extends AbstractNAV {
     -   public double computeNetAssetValue(final Map<String, Integer> stocks)
     -       throws InterruptedException, ExecutionException {
     -     final int numberOfCores = Runtime.getRuntime().availableProcessors();
     5     final double blockingCoefficient = 0.9;
     -     final int poolSize = (int)(numberOfCores / (1 - blockingCoefficient));
     -
     -     System.out.println("Number of Cores available is " + numberOfCores);
     -     System.out.println("Pool size is " + poolSize);
    10     final List<Callable<Double>> partitions =
```

```
      new ArrayList<Callable<Double>>();
    for(final String ticker : stocks.keySet()) {
      partitions.add(new Callable<Double>() {
        public Double call() throws Exception {
          return stocks.get(ticker) * YahooFinance.getPrice(ticker);
        }
      });
    }

    final ExecutorService executorPool =
      Executors.newFixedThreadPool(poolSize);
    final List<Future<Double>> valueOfStocks =
      executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);

    double netAssetValue = 0.0;
    for(final Future<Double> valueOfAStock : valueOfStocks)
      netAssetValue += valueOfAStock.get();

    executorPool.shutdown();
    return netAssetValue;
  }

  public static void main(final String[] args)
    throws ExecutionException, InterruptedException, IOException {
    new ConcurrentNAV().timeAndComputeValue();
  }
}
```

In the computeNetAssetValue() method we determine the thread pool size based on the presumed blocking coefficient and the number of cores (Runtime's availableProcessor() method gives that detail). We then place each part—to fetch the price for each ticker symbol—into the anonymous code block of the Callable interface. This interface provides a call() method that returns a value of the parameterized type of this interface (Double in the example). We then schedule these parts on the fixed-size pool using the invokeAll() method. The executor takes the responsibility of concurrently running as many of the parts as possible. If there are more divisions than the pool size, they get queued for their execution turn. Since the parts run concurrently and asynchronously, the dispatching main thread can't get the results right away. The invokeAll() method returns a collection of Future objects once all the scheduled tasks complete.[4] We request for the partial results from these objects and add them to the net asset value. Let's see how the concurrent version performed:

4. Use the CompletionService if you'd like to fetch the results as they complete and rather not wait for all tasks to finish.

• CLICK HERE to purchase this book now. discuss

```
Number of Cores available is 2
Pool size is 20
Your net asset value is $13,661,010.17
Time (seconds) taken 0.967484
```

In contrast to the sequential run, the concurrent run took less than a second. We can vary the number of threads in the pool by varying the presumed blocking coefficient and see whether the speed varies. We can also try different numbers of stocks as well and see the result and speed change between the sequential and concurrent versions.

Isolated Mutability

In this problem, the executor service pretty much eliminated any synchronization concerns—it allowed us to nicely delegate tasks and receive their results from a coordinating thread. The only mutable variable we have in the previous code is `netAssetValue`, which we defined on line 25. The only place where we mutate this variable is on line 27. This mutation happens only in one thread, the main thread—so we have only isolated mutability here and not shared mutability. Since there is no shared state, there is nothing to synchronize in this example. With the help of `Future`, we were able to safely send the result from the threads fetching the data to the main thread.

There's one limitation to the approach in this example. We're iterating through the `Future` objects in the loop on line 26. So, we request results from one part at a time, pretty much in the order we created/scheduled the divisions. Even if one of the later parts finishes first, we won't process its results until we process the results of parts before that. In this particular example, that may not be an issue. However, if we have quite a bit of computation to perform upon receiving the response, then we'd rather process results as they become available instead of waiting for all tasks to finish. We could use the JDK `CompletionService` for this. We'll revisit this concern and look at some alternate solutions later. Let's switch gears and analyze the speedup.

2.3 Speedup for the IO-Intensive App

The nature of IO-intensive applications allows for a greater degree of concurrency even when there are fewer cores. When blocked on an IO operation, we can switch to perform other tasks or request for other IO operations to be started. We estimated that on a two-core machine, about twenty threads would be reasonable for the stock total asset value application. Let's analyze the performance on a two-core processor for various numbers of threads—from one to forty. Since the total number of divisions is forty, it would not make any sense to create more threads than that. We can observe

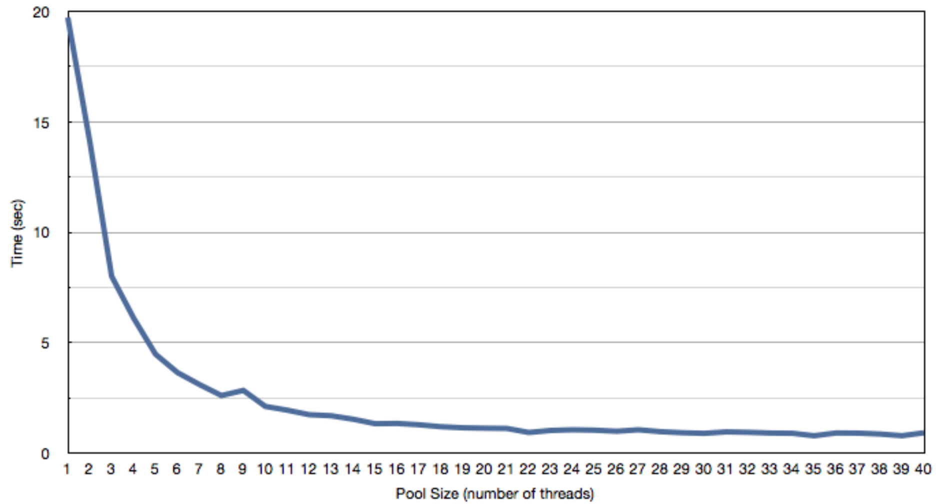


Figure 1—Speedup as the pool size is increased

the speedup as the number of threads is increased in Figure 1, *Speedup as the pool size is increased*, on page 15.

The curve begins to flatten right about twenty threads in the pool. This tells us that our estimate was decent and that having more threads beyond our estimate will not help.

This application is a perfect candidate for concurrency—the workload across the parts is about the same, and the large blocking, because of data request latency from the Web, lends really well to exploiting the threads. We were able to gain a greater speedup by increasing the number of threads. Not all problems, however, will lend themselves to speedup that way, as we'll see next.

• CLICK HERE to purchase this book now. discuss