Comparing the Performance of Implicit and Explicit Concurrency in Java

Dominic Rathbone

March 5, 2016

Introduction

In order to compare the effects of concurrency in Java on performance, A test program was created that compared various implementations of concurrency, implicit and explicit, to two baseline non-concurrent implementations. These implementations performed a number of mathematical operations on a set of integers derived from the user inputting a start and end integer at launch. These operations consisted of finding all the prime numbers in the set, finding all the factors of the end integer in the set, finding all the mersenne numbers in the set and finding all the perfect numbers in the set. Within this program, time recorded in nano seconds was used as a benchmark in order to compare these implementations to each other. Although there are other variables that could be used in conjunction that may give a more detailed result, time will give us clear and simple results to draw conclusions from. The time was taken before and after each implementation ran over the set and then each total run time was derived from them. In order to compare them easily, a ratio derived from dividing the total time of the baseline implementation by the total time of concurrent implementation is outputted to csv files.

To run the programs, I am using a ThinkPad laptop running Fedora with a quad-core 2.50GHz i5-2520M CPU and 8 gigabytes of RAM. as well as a Gaming PC running Windows 7 with a quad-core 3.50GHz i5 4690K and 8 gigabytes of RAM.

0.1 Non-concurrent Approach

The non-concurrent approach runs the operations sequentially, incrementing a counter every time the operation is successful. This is used as a baseline to compare the explicit concurrent approaches to.

0.2 Raw Multithreaded Approach

The first explicitly concurrent implementation uses raw threads, giving each operation a separate thread so they can be executed in parallel.

0.2.1 Performance

Fedora Laptop

Windows 7 PC

0.3 Threadpooled Approach

The second explicitly concurrent implementation

0.3.1 Performance

Fedora Laptop

Windows 7 PC

0.4 Serial Stream Approach

0.4.1 Performance

Fedora Laptop

Windows 7 PC

0.5 Parallel Stream Approach

0.5.1 Performance

Fedora Laptop

Windows 7 PC

Appendix A

Figure A.1: NonCurrentRunner.java

```
public void run() {
        for(int i = start; i <= end; i++) {
                if(!Calculator.isFactor(i,end)) {
                        factorCount++;
                }
                if(!Calculator.isPrimeNumber(i)) {
                        primeCount++;
                }
                if(!Calculator.isMersenne(i)) {
                        mersenneCount++;
                }
                if(!Calculator.isPerfectNumber(i)) {
                        perfectNumberCount++;
                }
        }
}
```

Appendix B

Figure B.1: RawMultithreadedRunner.java

```
public void run() {
        Thread threadA =
                new Thread(new ThreadA(start, end));
        Thread threadB =
                new Thread(new ThreadB(start, end));
        Thread threadC =
                new Thread(new ThreadC(start, end));
        Thread threadD =
                new Thread(new ThreadD(start, end));
        threadA.run();
        threadB.run();
        threadC.run();
        threadD.run();
}
```

Appendix C

Figure C.1: ThreadA.java

```
public void run() {
        for (int i = start; i <= end; i++) {
                if (!Calculator.isFactor(i, end)) {
                        factorCount++;
                }
        }
}
```

Appendix D

Figure D.1: ThreadPoolRunner.java

```
public void run() {
        ExecutorService executor =
                        Executors.newFixedThreadPool(6);
        executor.execute(new ThreadA(start, end));
        executor.execute(new ThreadB(start, end));
        executor.execute(new ThreadC(start, end));
        executor.execute(new ThreadD(start, end));
        executor.shutdown();
        while(!executor.isTerminated()) {
        }
}
```

Appendix E

Figure E.1: SerialStreamRunner.java

```
public void run() {
        Stream
        .iterate(start, i -> i++)
        .limit(end)
        .filter(i -> isFactor(i, end))
        .filter(i -> isPrime(i))
        .filter(i -> isMersenne(i))
        .filter(i -> isPerfect(i))
        .close();
}
```

Appendix F

Figure F.1: SerialStreamRunner.java

```
public void run() {
    Stream
    .iterate(start, i -> i++)
    .limit(end)
    .filter(i -> isFactor(i, end))
    .filter(i -> isPrime(i))
    .filter(i -> isMersenne(i))
    .filter(i -> isPerfect(i))
    .close();
}
```