PART A
PART 2
Comparing the Performance of Implicit and Explicit Concurrency in Java

Dominic Rathbone

March 19, 2016

Introduction

In order to compare the effects of concurrency on performance of a Java application, A test program was created that compared various implementations of concurrency, implicit and explicit, to a baseline non-concurrent implementation. These implementations performed a mathematical operation finding all the mersenne primes in a set of integers derived from the user inputting a start and end number at launch. Time (in nano seconds) was used as a benchmark in order to compare these implementations. The time was taken before and after each implementation ran and then the total run time was derived from them. In order to compare them with ease, 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, A ThinkPad laptop running Fedora with a quad-core 2.50GHz i5-2520M CPU and 8 gigabytes of RAM was used 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, running over the sequence of integers with a for statement and incrementing a counter every time a prime number is found. This is used as the baseline to compare the implicit and explicit concurrent approaches to.(see Appendix A)

0.2 Multi-threading

The first explicitly concurrent implementation uses raw threads to run the operation over the sequence of numbers. This was attempted in two ways. First of all, to make the test a better comparison, I produced an implementation that spawned a new thread for every number in the sequence (see Appendix B, figure 1), similar to how a parallel stream would split a stream of numbers into multiple parallel sub-streams. On testing this implementation, it was revealed that this would not be an efficient way to approach concurrency because as the sequence of numbers became larger, it would become extremely slow (finishing minutes where other approaches finished in seconds). This was due to the lack of thread management such as a queue or limit to how many threads should be started at once meaning the machine bottlenecked by a flood of threads. As a result of this, the implementation was re-factored and it was decided that the best way to approach this was to split the sequence of numbers down into smaller sequences using the number of cores with one subsequence per thread per core (see Appendix B, figure 2). This would mean that although each CPU core had to process with a larger sequence of numbers per thread, the number of threads and the overhead associated with them was lower making the time drastically faster.

0.3 Thread Pooling

The second explicitly concurrent implementation was through the use of thread pools and futures. A thread pool is a group of pre-defined threads that can be used to run tasks. In this approach, a task is submitted to the thread pool for every number in the sequence. This is similar to the raw multi-threaded implementation but the thread pool can queue the threads so they do not start at the same time and bottleneck the machine as mentioned in the previous section. The results of this task is then retrieved through a ”Future”, an object that represents the results from the submission of the task (see Appendix C).

0.4 Streaming

This approach takes the sequence of number and processes it as a stream, applying intermediate operations on each element in the stream until it reaches an terminal operation which ends the set of operations. This can be done sequentially with a serial stream (see Appendix D) or concurrently with a parallel stream (see Appendix E). The result set includes both serial and parallel stream in order to compare how they perform.

0.5 Data Analysis

From the data (Appendix F), we can see that running the program on the desktop PC gave better results on average compared with running them on the laptop with the highest ratio for each concurrent implementation running on PC being 1.5x to 2x faster than on the laptop and the average ratio being 1.5x faster on the PC than on the laptop. This implies that all of these concurrent implementations do scale with hardware and that, vice versa, worse hardware will bottleneck concurrent applications. However, there are other variables that could have affected the performance such as other processes using the CPU during the period the program was running or the different operating system that the tests were being ran on (Windows 7 & Fedora 22).

Across the implementations, on both PC and laptop, it can be seen that on average, The raw multi-threaded approach and the parallel stream were faster compared to the baseline than the thread pool approach especially as the size of data set got larger. However, the graphs (Appendix G) indicate that on both machines, as the end value got larger, the thread pool can be seen to improve dramatically in performance, in particular when the end value reached 801million. This could imply that the programs were not ran over a data set with a large enough end value and that the thread pool may actually perform just as well or better as the numbers increased past this point. This is especially worth noting as the highest ratio (1:3.7972339477 on PC, 1:2.0458180569 on laptop) from the thread pool data found at the 800-801 million data set is higher than the highest ratio from the parallel stream approach (1:3.5239193257 on PC, 1:2.0271960552 on laptop). On the other hand, if we look at the graph mapping the size of the data set to performance, the thread pool approach seems to peak at a size of 1,000,000 to 3,000,000 and then drop as the size increases past this point implying that although it may perform at larger numbers, performance might decrease as the distance between the start value and the end value gets larger.

It can be seen from the graphs (Appendix G) that, across all concurrent implementations and machines, as the end value of the data sets increased, the performance initially increased until a plateau was reached at around the size of 200000. The increase could be because as the numbers got larger, the cost of the overhead of running the tasks concurrently in each implementation was outweighed by the benefits of calculating these tasks in this manner. The plateau implies that the software reached a point where hardware limitations such as the number of cores started to bottleneck the program.

One unusual result was how the performance of the multi-threaded approach spiked twice very early on in the graph which is surprising as the cost of using threads on data sets this small usually outweighs the benefit. This occurred on both the desktop PC and the laptop results which implies that it isn't caused by something machine-specific.

Overall, the results reflected what was expected from the tests with the exception of the thread pool approach, of which I expected to be faster than the multi-threaded approach due it's thread management capabilities. However, due to how I re-factored the multi-threaded approach to split the sequence of numbers differently, it may not be fair to compare the two. If I were to leave the multi-threaded implementation to run the operations in a similar manner to the thread-pool, it would most likely vastly be out-performed.

Appendix A

Figure A.1: Non-concurrent

```
public static void nonConcurrent() {
        for ( long n = START; n < END; n++ ) {
                int primeCount = 0;
                if (Calculator.isMersennePrime(n)) {
                        primeCount++;
                }
        }
}
```

Appendix B

Figure B.1: Multi-threaded 1

```
public static void multiThreaded() {
        List<Thread> threads = new ArrayList<>();
                for (long n = START; n < END; n++) {
                        Thread thread = new Thread(
                                new RunnableA(n)
                        );
                        thread.start();
                        threads.add(thread);
                }
                for(Thread thread: threads) {
                        try {
                                thread.join();
                        } catch (InterruptedException e){
                                e.printStackTrace();
                        }
                }
}
```

Figure B.2: Multi-threaded 2

```
public static void multiThreaded2() {

        List<Thread> threads = new ArrayList<>();
        long subsequence = (END - START) / CORES;

        for ( long n = START; n < END; n=n+subsequence) {
                Thread thread = new Thread(
                        new RunnableB(n, n+subsequence)
                );
                thread.start();
                threads.add(thread);
        }

        for(Thread thread: threads) {
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Appendix C

Figure C.1: Thread Pool

```
public static void threadPool() {
        ExecutorService executorService =
                Executors.newFixedThreadPool(CORES);
        ArrayList<Future<Integer>> futures =
                new ArrayList<>();

        for ( long n = START; n < END; n++ ) {
                final long number = n;
                Future<Integer> future =
                executorService.submit(() -> {
                  if(Calculator.isMersennePrime(number)) {
                          return 1;
                  }
                  else {
                          return 0;
                  }
                }
                );
                futures.add(future);
        }

        int primeCount = 0;
        for (Future<Integer> future : futures) {
                try {
                        primeCount += future.get();
                } catch (Exception e) {
                }
        }
        executorService.shutdown();
}
```

Appendix D

Figure D.1: Serial Stream

```
public static void serialStream() {
        long primeCount = Stream
        .iterate(START, n -> n + 1)
        .limit(END-START)
        .filter( n -> Calculator.isMersennePrime(n) )
        .count();
}
```

Appendix E

Figure E.1: Parallel Stream

```
public static void parallelStream() {
        long primeCount = Stream
        .iterate(START, n -> n + 1)
        .limit(END-START)
        .parallel()
        .filter(n -> Calculator.isMersennePrime(n) )
        .count();
}
```

Appendix F

results_4_CORES_PC_BY_END_VALUE

START VALUE	END VALUE	NC/MT ratio	NC/TP ratio	NC/SS ratio	NC/PS ratio	RANGE
1000	2000	3.377639498	0.31913372	0.027929964	0.034146773	1000
1000	2000	0.320807279	0.055333735	0.286081339	0.252846387	1000
1000	2000	0.239644043	0.068624413	0.357876721	0.172952854	1000
1000	2000	0.196722643	0.067835434	0.394739209	0.194176127	1000
1000	2000	0.225326922	0.073045897	0.398684603	0.200000565	1000
1000	2000	1.758723902	0.253959165	0.024393171	0.023739355	1000
1000	2000	0.217272688	0.041493072	0.250262166	0.165512344	1000
1000	2000	0.220710359	0.051581958	0.374426602	0.19209725	1000
1000	2000	0.170696852	0.060717377	0.328892813	0.163466585	1000
1000	2000	0.210390246	0.074173599	0.425127366	0.220730944	1000
10000	20000	0.858815407	0.158652262	0.725508847	0.51270343	10000
10000	20000	1.205827301	0.173652192	0.823844341	0.465417258	10000
10000	20000	1.176961127	0.262230756	0.816759772	0.747987943	10000
10000	20000	1.151818063	0.196830896	0.825228021	1.050290563	10000
10000	20000	1.188513651	0.269173537	0.859399262	0.878229009	10000
10000	20000	0.766828061	0.13161994	0.667895594	0.566473385	10000
10000	20000	1.181371339	0.253831024	0.805998556	0.484615564	10000
10000	20000	1.859411256	0.245705539	0.831074158	0.799089241	10000
10000	20000	1.207935607	0.188525057	0.82735425	1.078820341	10000
10000	20000	1.19423832	0.284880282	0.852876299	0.936588513	10000
100000	200000	2.340828048	0.506817026	0.9082702	1.865600518	100000
100000	200000	2.125376932	0.416007445	0.963233115	2.219974061	100000
100000	200000	3.293454721	0.595493681	0.957305169	2.259292598	100000
100000	200000	3.368788592	0.615754195	0.957900049	2.594866714	100000
100000	200000	3.231993278	0.624263085	0.966778684	2.051144435	100000
100000	200000	2.258560136	0.513285156	0.926996047	2.006472467	100000
100000	200000	3.415948165	0.434204296	0.977496539	1.64506166	100000
100000	200000	3.139018057	0.612744375	0.958686746	2.246692809	100000
100000	200000	3.51202543	0.644250985	0.991885105	2.71182824	100000
100000	200000	3.214072594	0.62013467	0.951310813	2.217843599	100000
1000000	1300000	4.451915331	1.278310212	1.006379802	1.322046076	300000
1000000	1300000	3.856927662	1.260232203	0.970579024	3.523919326	300000
1000000	1300000	3.792950079	1.159014021	0.950371928	3.258594083	300000
1000000	1300000	3.770619274	1.223875962	0.9181477	3.355876674	300000
1000000	1300000	3.794271987	1.382822903	0.959746074	3.448348137	300000
1000000	2000000	3.514506636	1.421082941	0.985529534	3.073749118	1000000
1000000	2000000	3.519729814	1.671682961	0.967841018	3.205534829	1000000
1000000	2000000	3.624065796	0.745356153	0.978514607	3.192181753	1000000
1000000	2000000	3.614826372	1.572081096	0.955582479	3.203334201	1000000
1000000	2000000	3.619815149	1.523739953	0.976732052	3.148937508	1000000
1000000	2000000	3.517761987	1.312032542	0.982016502	3.079629016	1000000
1000000	2000000	3.588292519	1.587881331	0.988058455	3.078592823	1000000
1000000	2000000	3.616462549	1.408812218	0.980326377	3.17659234	1000000
1000000	2000000	3.582887706	0.749203995	0.97802057	3.170586645	1000000
1000000	2000000	3.633200606	1.664774504	0.983058502	3.23337179	1000000
4000000	16000000	3.362055883	1.046243234	0.993858044	2.034823663	12000000
4000000	16000000	3.382761937	1.334551493	0.9883842	3.118080474	12000000
4000000	16000000	3.412241354	1.735529724	0.958752682	3.065004549	12000000
4000000	16000000	3.243706726	1.871174476	0.952584622	3.028928073	12000000
4000000	16000000	3.469829834	2.470716633	0.946710912	3.069480705	12000000
10000000	20000000	3.748113149	1.747108969	0.999592546	3.266074489	10000000
10000000	20000000	3.695950777	2.177803496	0.995090801	2.470590206	10000000

10000000	20000000	3.692996818	1.917985992	0.98914144	2.476299678	10000000
10000000	20000000	3.685017147	1.634481984	0.999603464	3.160641051	10000000
10000000	20000000	3.141016189	1.471266483	1.006922844	3.161235007	10000000
10000000	20000000	3.723301464	1.86687083	0.989056198	3.172549963	10000000
10000000	20000000	3.716738113	2.277855382	0.994316709	3.129259425	10000000
10000000	20000000	3.645727715	2.403086301	0.994768397	2.911261845	10000000
10000000	20000000	3.365285863	2.375225615	0.991145406	3.062920633	10000000
10000000	20000000	3.725905115	2.305160755	0.986125472	3.025711226	10000000
800000000	801000000	3.362653602	3.267070376	0.991573219	3.246680993	1000000
800000000	801000000	3.623860248	3.452837263	0.993757194	3.257731374	1000000
800000000	801000000	3.643522237	3.643523353	1.024524142	3.297184018	1000000
800000000	801000000	3.687123738	3.750505077	1.027722113	3.277188692	1000000
800000000	801000000	3.716744776	3.797233948	1.030373262	3.354370366	1000000

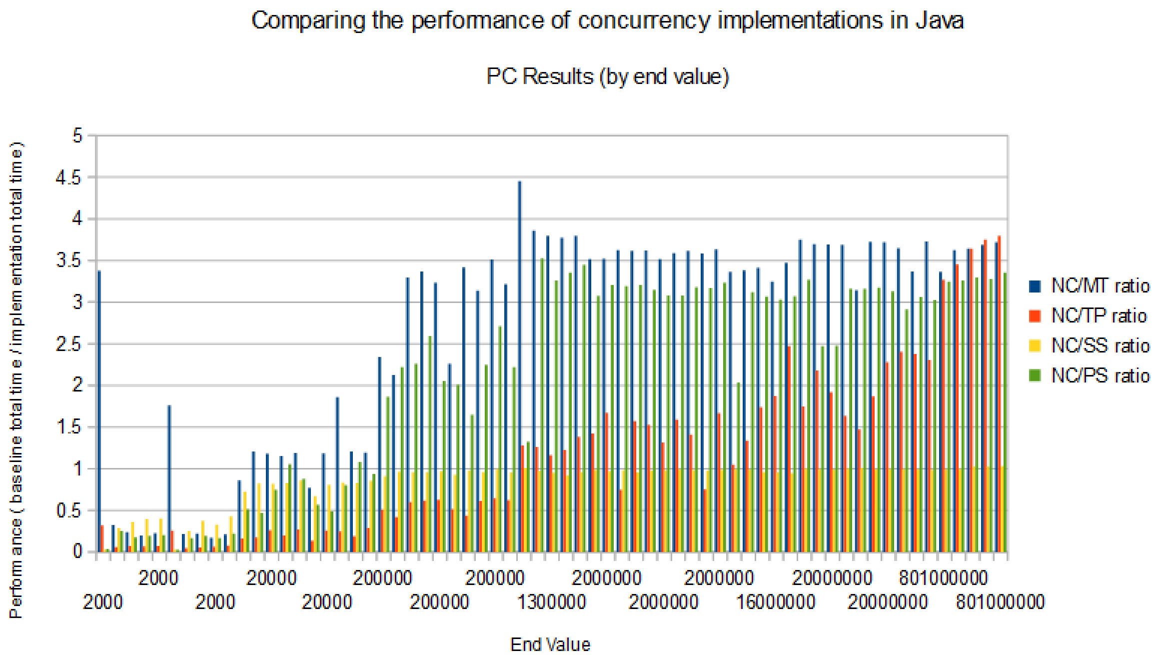
results_4_CORES_LAPTOP_BY_END_VALUE

START VALUE	END VALUE	NC/MT ratio	NC/TP ratio	NC/SS ratio	NC/PS ratio	RANGE
1000	2000	1.822415532	0.174783803	0.021748569	0.1860857	1000
1000	2000	0.161616839	0.024289526	0.253046546	0.119190818	1000
1000	2000	0.048011791	0.04524044	0.325850436	0.090595202	1000
1000	2000	0.403661601	0.065935929	0.582434319	0.154057565	1000
1000	2000	0.284183221	0.06703164	0.552081013	0.159012571	1000
1000	2000	1.842330849	0.160027263	0.019974901	0.159624162	1000
1000	2000	0.177369389	0.031642014	0.310330343	0.098413259	1000
1000	2000	0.012354499	0.0307123	0.276183797	0.100820143	1000
1000	2000	0.077390854	0.066994761	0.391656298	0.164510317	1000
1000	2000	0.203584745	0.03672512	0.49585795	0.1425688	1000
10000	20000	0.234928128	0.100118398	0.729529159	0.448235394	10000
10000	20000	0.570385742	0.151337512	0.984669945	0.634057303	10000
10000	20000	0.319487332	0.24590822	0.625115128	0.471049947	10000
10000	20000	1.137688373	0.189088368	0.80783934	0.746968622	10000
10000	20000	0.950581139	0.177498382	0.822928318	0.843591044	10000
10000	20000	0.509015271	0.098484672	0.450664007	0.313595908	10000
10000	20000	0.978315001	0.185763271	0.553286727	0.38131843	10000
10000	20000	0.278352671	0.259175352	0.579025524	0.387977223	10000
10000	20000	1.158372939	0.159446671	0.830768952	0.881910015	10000
10000	20000	0.9439115	0.143231888	0.855080441	0.874236673	10000
100000	200000	1.771160604	0.385661468	0.946951505	1.001947547	100000
100000	200000	1.479750051	0.428212203	0.90944275	0.898461912	100000
100000	200000	1.283563869	0.404342675	0.975244854	1.580723771	100000
100000	200000	1.705189941	0.389480979	0.972299909	1.64552861	100000
100000	200000	1.976323106	0.459602293	0.959332167	1.620017516	100000
100000	200000	1.595393784	0.513201657	0.873398782	1.069756077	100000
100000	200000	1.664986356	0.345129971	0.87739217	0.928963124	100000
100000	200000	1.467217539	0.50376251	1.003307829	1.646249868	100000
100000	200000	1.880497996	0.38833524	0.964506042	1.565578952	100000
100000	200000	1.995171915	0.45527218	0.975070727	1.529786788	100000
1000000	2000000	2.089357622	0.791744079	0.963785758	1.796101601	1000000
1000000	2000000	1.988806744	0.482402106	0.883167699	1.755362029	1000000
1000000	2000000	2.050816105	0.741354332	0.981150038	1.785105642	1000000
1000000	2000000	2.081661363	0.960609993	0.959895347	1.795616736	1000000
1000000	2000000	1.898397087	0.901628952	0.969452616	1.727401005	1000000
1000000	2000000	2.09032314	0.851776595	0.971019988	1.865149783	1000000
1000000	2000000	2.046910691	0.463840128	0.956443561	1.868196816	1000000
1000000	2000000	2.077055987	0.720316127	0.967484725	1.696934362	1000000
1000000	2000000	2.040108794	0.841348667	0.968571073	1.904896911	1000000
1000000	2000000	2.074644999	0.788702468	0.947120161	1.732486578	1000000
10000000	13000000	2.126506745	0.952776447	0.98156175	1.295640381	3000000
10000000	13000000	1.983727574	0.936264658	0.951409062	1.878472587	3000000
10000000	13000000	2.060761698	1.384711071	0.979501647	1.833994618	3000000
10000000	13000000	1.971154365	1.37110004	0.954518667	1.836056547	3000000
10000000	13000000	2.120054759	1.320480343	0.984174719	1.857594647	3000000
4000000	16000000	2.069670227	0.736040106	0.970988022	1.22231773	12000000
4000000	16000000	1.866702102	0.782743563	0.93249456	1.728418534	12000000
4000000	16000000	1.914680848	0.755840154	0.960232511	1.515119466	12000000
4000000	16000000	2.015967698	1.080844308	1.0379565	2.027196055	12000000
4000000	16000000	1.93975733	0.880806754	0.967207957	1.787200522	12000000
10000000	20000000	1.984645582	0.896206526	0.977812771	1.645463362	10000000
10000000	20000000	1.904503518	0.886975958	0.959679234	1.844449905	10000000

10000000	20000000	2.032640562	1.05033448	0.975789427	1.878804544	10000000
10000000	20000000	1.977990641	0.946475655	0.975593993	1.787809845	10000000
10000000	20000000	2.106130586	1.127567337	0.970219037	1.861740716	10000000
10000000	20000000	2.047182357	0.893938868	0.983768078	1.587451144	10000000
10000000	20000000	1.996461115	0.981548645	0.966075405	1.828673122	10000000
10000000	20000000	2.094017136	1.131197097	0.968992706	1.853934805	10000000
10000000	20000000	2.030319183	1.145986953	0.978175409	1.800510511	10000000
10000000	20000000	2.091426473	1.086843941	0.97335341	1.567636952	10000000
800000000	801000000	2.158586995	1.900791289	1.010628296	1.986966865	1000000
800000000	801000000	2.026315654	1.935729615	0.933755046	1.833623535	1000000
800000000	801000000	2.100828807	2.007524164	0.985084759	1.967674223	1000000
800000000	801000000	2.110788815	2.045818057	0.989801464	1.942106473	1000000
800000000	801000000	2.107337597	1.993997269	0.984795236	1.941794331	1000000

Appendix G

Figure G.1: PC Results



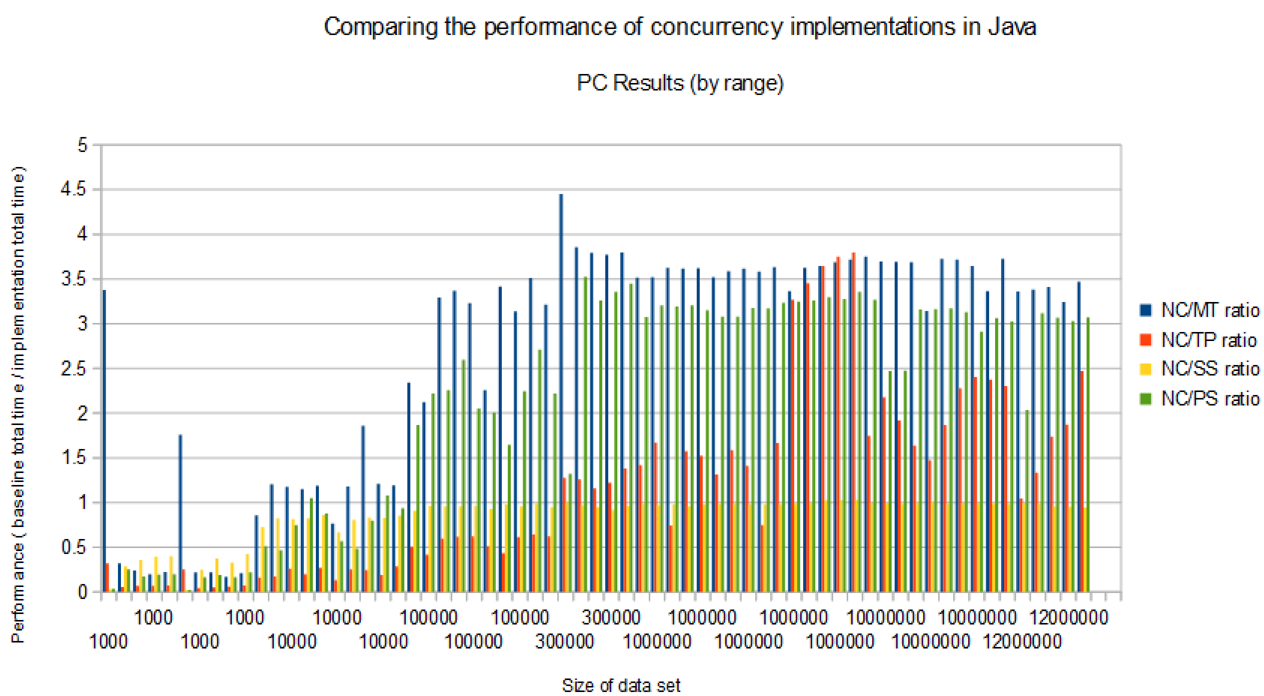


Figure G.2: Laptop Results