Oregon State University

CS 311 HW5

Threads and shared memory

Peter Rindal
2/25/2013

Program Design

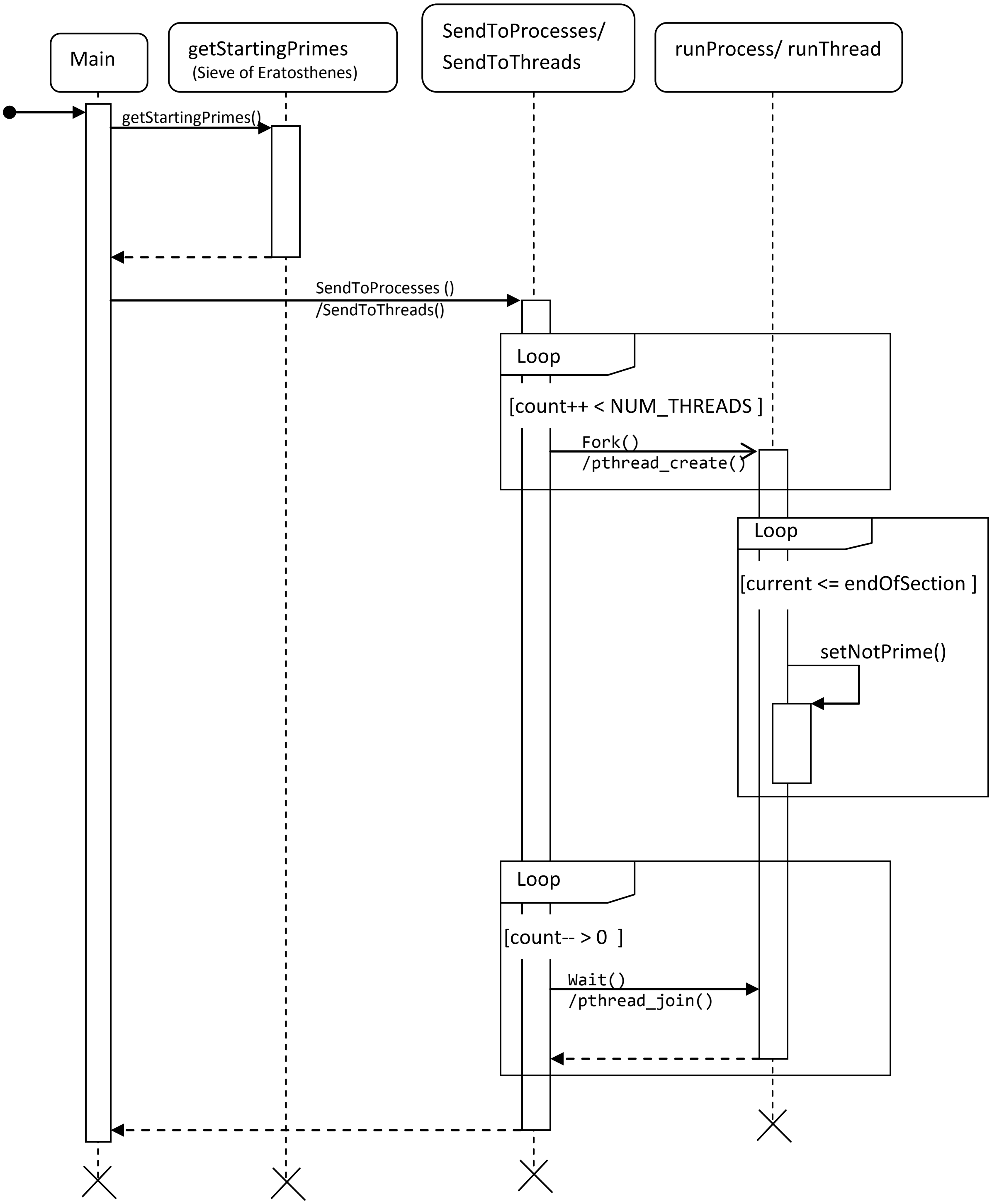
My program is modeled after the Sieve of Eratosthenes and is broken up into two main parts. The first part is the Sieve of Eratosthenes algorithm in its full. However I only use it to calculate the primes that fall within the range $\{0, 1, 2, \dots , n^{\frac{1}{2}}\}$, where n is the upper bound on the overall range. Because the square root is so much smaller than the overall range this takes next to no time to calculate and is efficient.

The second part of my program takes advantage of parallel processing. In this part I find the remaining primes in the range $\{n^{\frac{1}{2}} + 1, n^{\frac{1}{2}} + 2, \dots , n - 1 , n\}$. My program does this by dividing this range up between all of the worker threads/processes. So if my program was told to use 10 threads/processes then the first tenth of the range will be given to the first thread/process. The second tenth will be given to the second thread/process and so on. In addition to knowing what section a thread/process will be working on, it knows all of the primes in the range $\{0, 1, 2, \dots , n^{\frac{1}{2}}\}$. This allows each thread/process to loop through its own section and eliminate all of the multiples of the primes that were found in the range $\{0, 1, 2, \dots , n^{\frac{1}{2}}\}$. Once a thread/process has eliminated all of the multiples in its section it terminates and waits to be reaped.

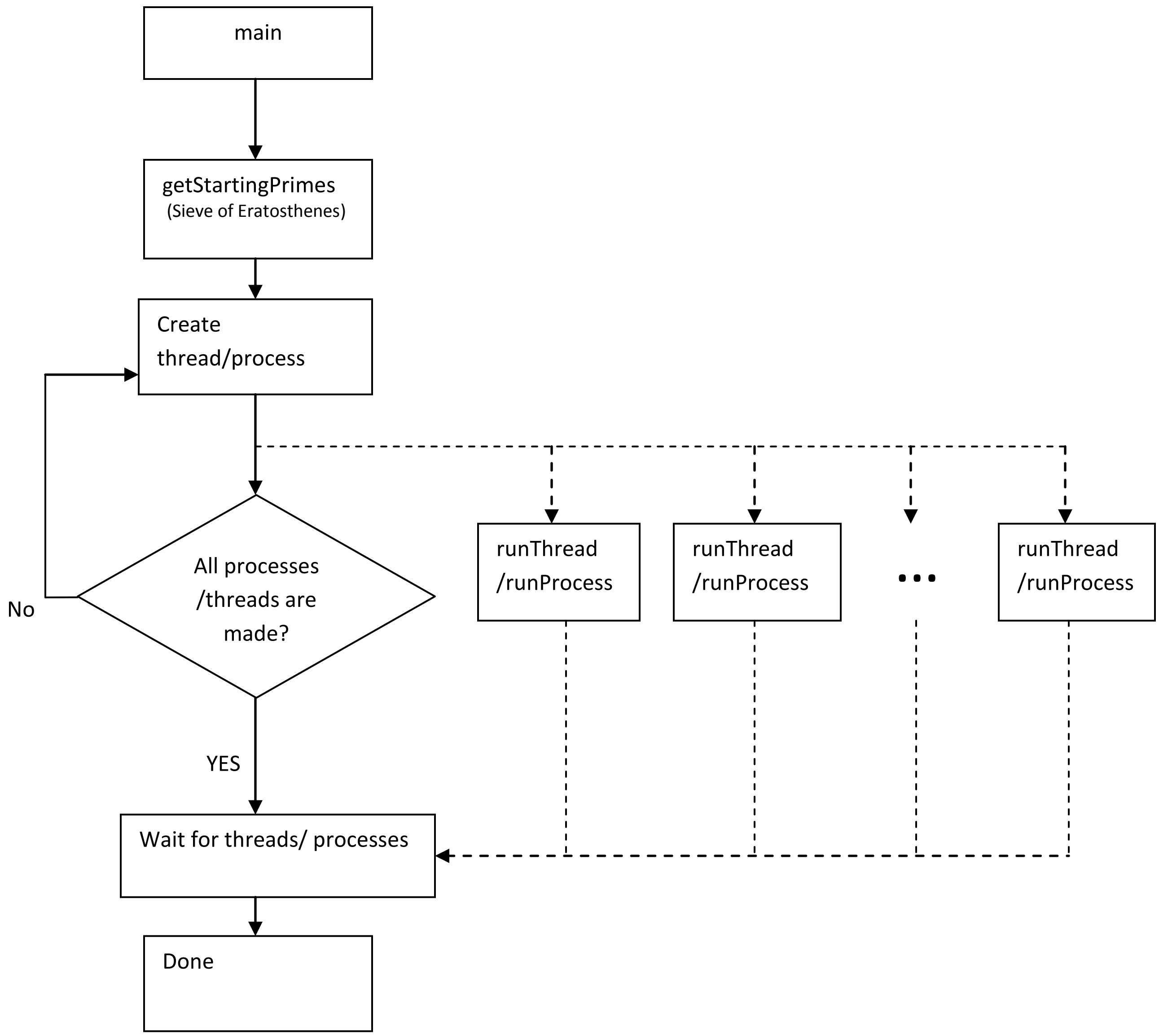
To avoid having any race conditions between any of threads/processes some extra care has to be taken about where to have one section end and the next start. This is only an issue because my program is using a bitmap to represent whether a number is prime or not. The C programming language has no native type that is made up of a single bit. This means that to create a bitmap I had to use a data type that is several bits long and hence it is possible for two threads/processes to attempt to write to it at the same time if the sections of these threads/processes meet in the middle of this data type. If this happens only one of these writes will persist and a prime will go missed. To avoid this race condition my program makes sure that every section starts and ends at an edge of a data type. One of my programs uses the char data type to make the bitmap. A char is made up of 8 bits and therefore all of the sections have to start on multiples of 8 and end on multiples of 8 -1. By doing this it guarantees that two threads/processes will never be working on the same piece of data.

One performance improving feature I added was to have a thread/process never work on more than one MB of the bitmap at a time. I believe this increases performance because the computer can hold that much in its fast memory and does not have to do as much waiting for the data in the slower memory. My programs accomplish this by having the threads/processes loop through its section in increments of 1 million numbers at a time.

Sequence Diagram



Flow Chart

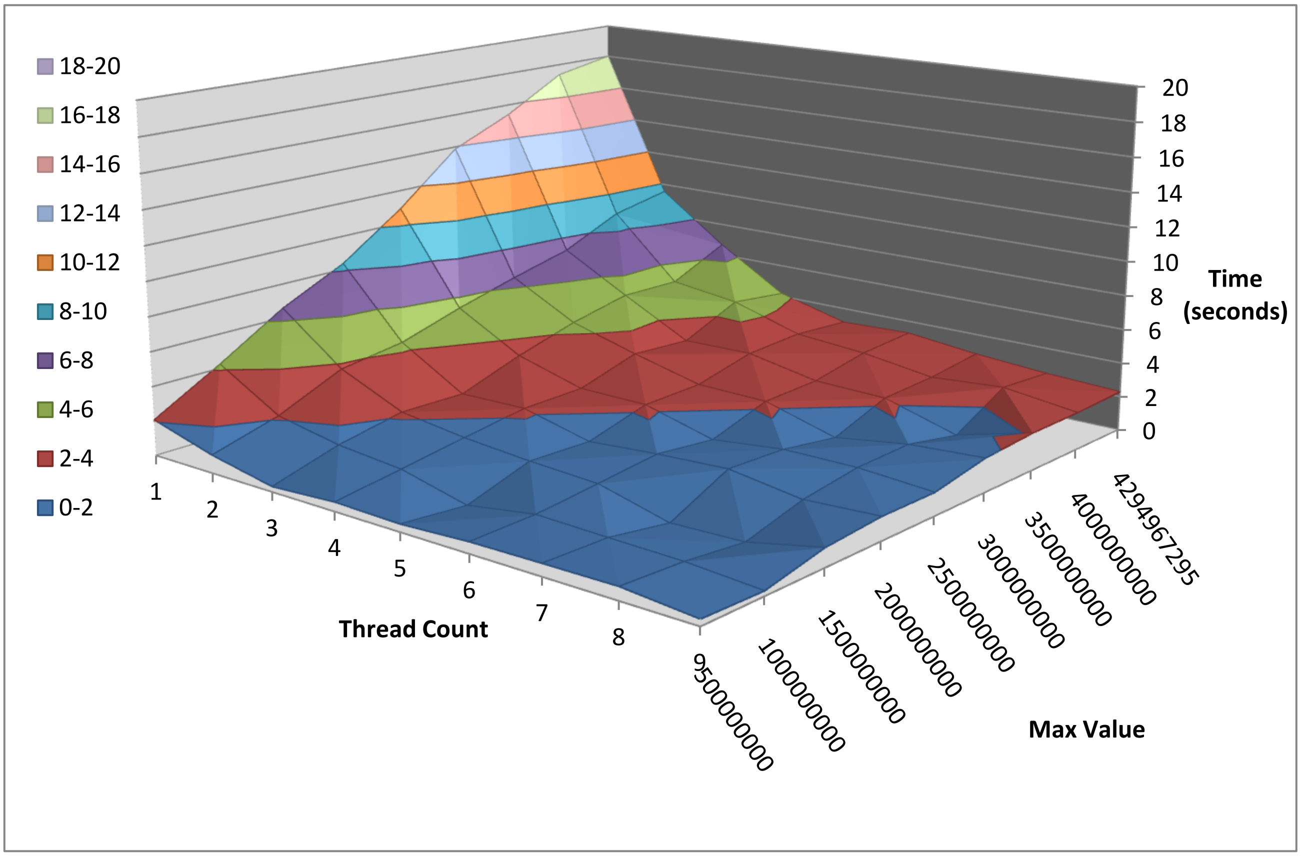


Work Log

This project went pretty smoothly. The first work I did on it was on Thursday the 21st of February. I worked on it for probably 4-5 hours and got 90% of the multi threaded code written. The next day I spent another 4-5 hours finishing up the code and debugging. On Saturday I re wrote my program to use processes and added the feature where each thread/process will only look at 1 million numbers at a time. This too another 4 to 5 hours. On Sunday the 24th I created a python script to run my program a bunch of time with a variety of inputs and recorded the running times of the program. I used this data to create the timing plots in the Timing section.

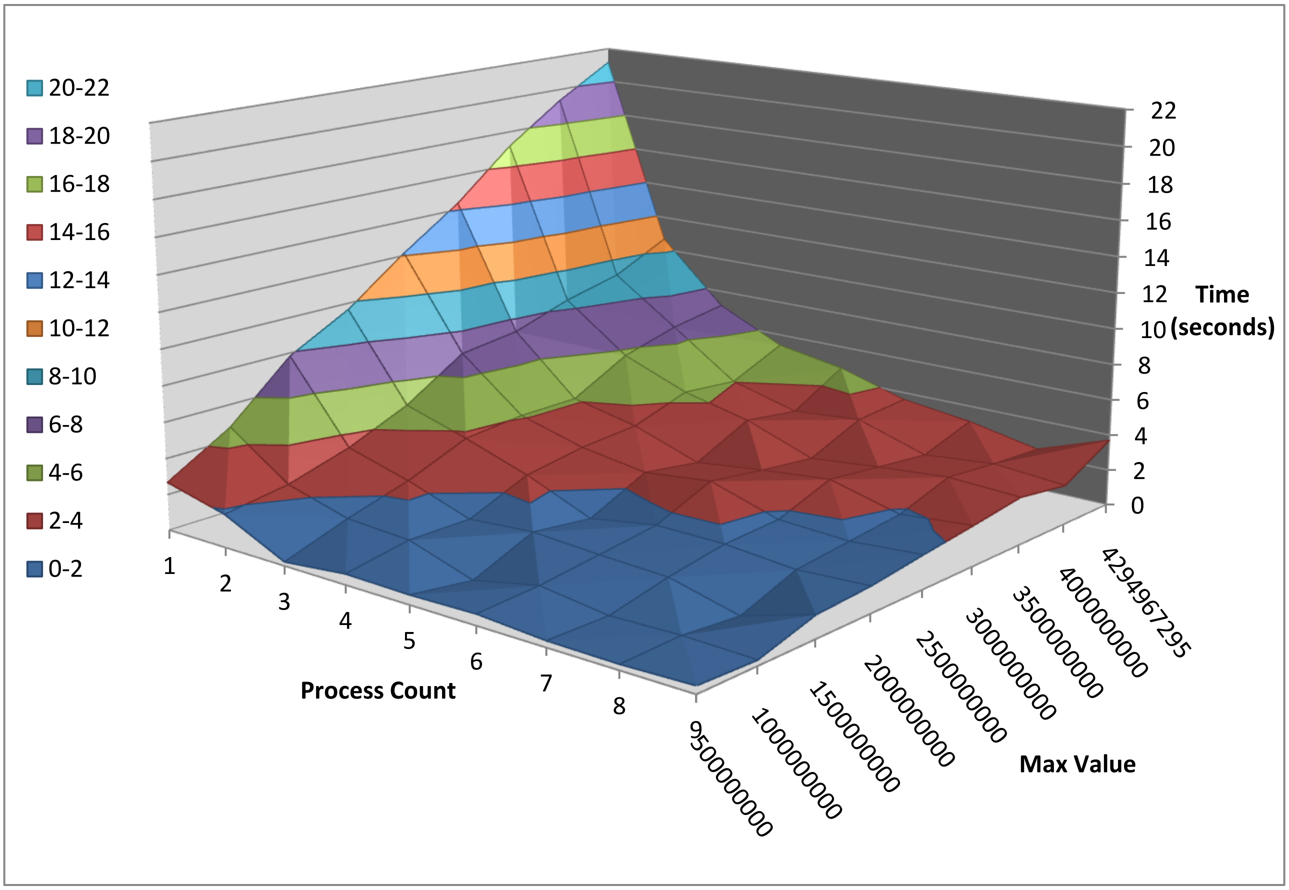
Timing

Multi-Threaded

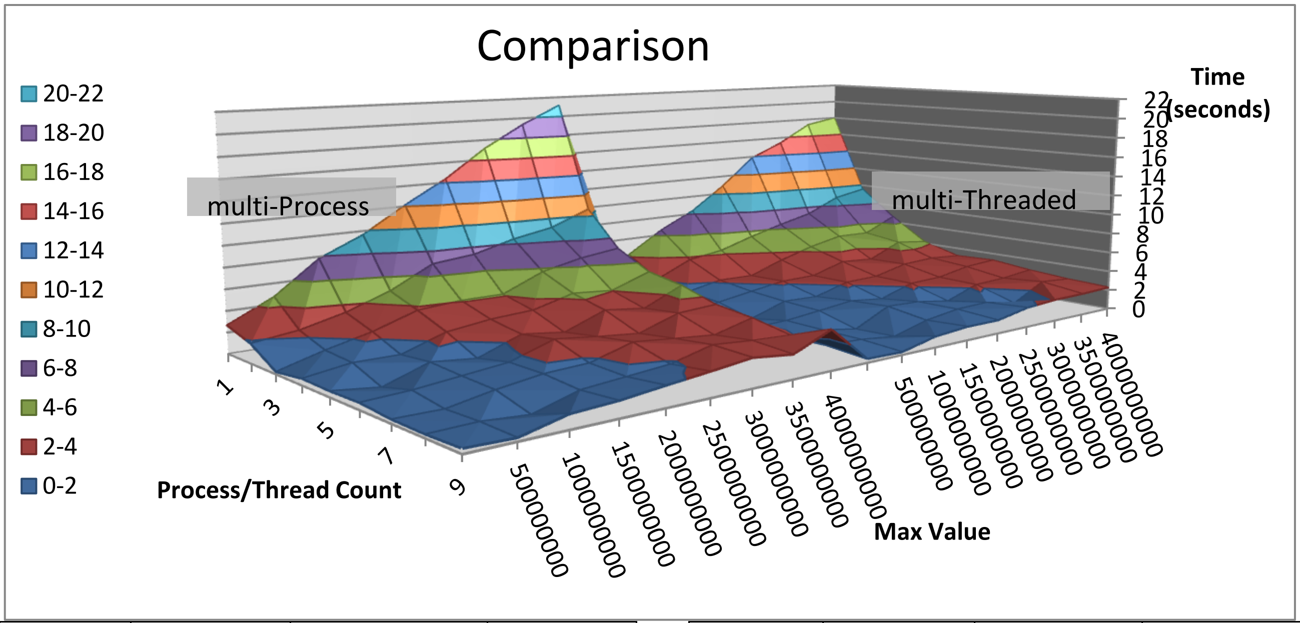


max value	TIMES								
4294967295	18.002	9.52387	6.64558	4.1757	3.11315	3.2469	2.78819	2.47417	2.28185
4000000000	17.3492	8.83531	5.09155	4.56821	3.34221	3.01149	2.41018	2.71244	2.16798
3500000000	15.3876	7.29208	5.14721	3.08262	3.15041	2.3664	2.25516	1.02191	2.14808
3000000000	13.8724	6.42294	4.39336	3.29609	2.31974	2.24036	1.07272	1.68783	1.93177
2500000000	10.8795	5.44975	3.35455	2.2558	2.22098	1.14594	1.59464	1.5995	1.30215
2000000000	8.34722	4.32906	2.10433	2.18313	1.2272	1.52707	1.72758	1.11401	1.31737
1500000000	6.56307	3.77714	2.15454	1.64473	1.32233	1.11121	0.05656	0.16775	1.04248
1000000000	4.24292	2.1241	1.42154	1.06677	0.12201	0.73365	0.62938	0.55018	0.26853
500000000	2.08316	1.04332	0.30124	0.52617	0.43804	0.63898	0.69083	0.72382	0.3606
	1	2	3	4	5	6	7	8	9
	thread count								

Multi-Process



max value		TIMES								
4294967295		21.1617	10.6277	7.10886	5.35072	4.69979	3.57694	3.08529	2.30529	3.68752
4000000000		19.3342	9.13394	6.60595	4.02967	3.02442	3.35366	2.14075	2.50277	2.18099
3500000000		17.1417	8.40387	5.75383	4.33232	3.47937	2.10588	2.49225	2.81325	2.62495
3000000000		14.3718	7.34473	4.09377	3.30911	2.03157	2.48385	2.86991	1.86401	2.21173
2500000000		12.1334	6.91923	4.07649	3.08145	2.53307	2.94355	1.76454	1.55476	1.83554
2000000000		9.64138	4.83883	3.76195	2.55925	1.0462	1.63483	1.59607	1.23876	1.42467
1500000000		7.82681	3.39845	2.41113	1.19158	1.45275	1.21501	1.04678	0.07643	1.2241
1000000000		4.73219	2.37598	1.41016	1.2047	0.04061	0.80324	0.30167	0.39158	0.28914
500000000		2.67597	1.8335	0.21686	0.58837	0.52272	0.60262	0.34272	0.3019	0.4133
		1	2	3	4	5	6	7	8	9
		process count								



number of threads/ Processes	max value	Thread data (seconds)	process data (seconds)
1	4294967295	18.002047	21.16169
1	4000000000	17.349195	19.334202
1	3500000000	15.38764	17.141674
1	3000000000	13.872369	14.371757
1	2500000000	10.879483	12.133395
1	2000000000	8.347216	9.641375
1	1500000000	6.563066	7.826805
1	1000000000	4.242921	4.732189
1	500000000	2.083155	2.675969
2	4294967295	9.523866	10.627746
2	4000000000	8.835308	9.133939
2	3500000000	7.292079	8.403873
2	3000000000	6.422942	7.344729
2	2500000000	5.449752	6.919232
2	2000000000	4.329064	4.838829
2	1500000000	3.77714	3.398451
2	1000000000	2.124095	2.375977
2	500000000	1.043321	1.833504
3	4294967295	6.645576	7.10886
3	4000000000	5.091554	6.605947
3	3500000000	5.147212	5.753828
3	3000000000	4.393361	4.093766
3	2500000000	3.354554	4.076489
3	2000000000	2.104326	3.761945
3	1500000000	2.154543	2.411126
3	1000000000	1.421544	1.410158
3	500000000	0.301244	0.216857
4	4294967295	4.175702	5.350718
4	4000000000	4.568213	4.029674
4	3500000000	3.08262	4.33232
4	3000000000	3.296085	3.309109
4	2500000000	2.255795	3.08145
4	2000000000	2.183128	2.559249
4	1500000000	1.644732	1.191584
4	1000000000	1.066774	1.2047
4	500000000	0.526168	0.588369
5	4294967295	3.113152	4.699786
5	4000000000	3.342213	3.024423
5	3500000000	3.150408	3.479365
5	3000000000	2.319736	2.031573
5	2500000000	2.220984	2.533069
5	2000000000	1.227198	1.046202
5	1500000000	1.322329	1.45275
5	1000000000	0.12201	0.040611
5	500000000	0.438037	0.522723
6	4294967295	3.246904	3.576944
6	4000000000	3.011489	3.353658
6	3500000000	2.366399	2.105876
6	3000000000	2.240357	2.483845
6	2500000000	1.145938	2.943549
6	2000000000	1.527072	1.634827
6	1500000000	1.11121	1.215008
6	1000000000	0.733649	0.803239
6	500000000	0.638976	0.602623
7	4294967295	2.788188	3.085292
7	4000000000	2.410179	2.140751
7	3500000000	2.255163	2.492245
7	3000000000	1.072718	2.869911
7	2500000000	1.594641	1.764543
7	2000000000	1.727575	1.596074
7	1500000000	0.056561	1.046779
7	1000000000	0.629384	0.301667
7	500000000	0.69083	0.342723
8	4294967295	2.474165	2.305286
8	4000000000	2.712439	2.502771
8	3500000000	1.021913	2.813247
8	3000000000	1.687827	1.864011
8	2500000000	1.599498	1.554759
8	2000000000	1.114008	1.238762
8	1500000000	0.167751	0.076429
8	1000000000	0.550178	0.391578
8	500000000	0.723821	0.301897
9	4294967295	2.281847	3.68752
9	4000000000	2.167977	2.180993
9	3500000000	2.148077	2.624952
9	3000000000	1.931765	2.211734
9	2500000000	1.302151	1.835537
9	2000000000	1.31737	1.424666
9	1500000000	1.042476	1.224098
9	1000000000	0.268533	0.289144
9	500000000	0.360603	0.413299

Discussion

From the moment I read there was extra credit for beating the time of 45 seconds I started thinking about performance and an architecture that would support this goal. In the text book they talk a good amount how mutexes and semaphores can slow down your program if your program is waiting on them a lot. Based on this idea and some discussions with a fellow student I decided to use a program design that doesn't use a single mutex or semaphore by not having any critical sections.

I achieve this by having every thread/process work on its own section of numbers. This allows every thread/process to be completely independent and will never have to wait for someone else to finish their operations. The parent thread will generate the primes in the range $\{0, 1, 2, \ldots, n^{\frac{1}{2}}\}$. From these primes each child thread/process will eliminate all of the prime multiples in its own section, leaving only primes untouched.

The threaded version of my program has found all of the primes smaller that 2^{32} in 2.03 seconds. Considering to get extra credit I only needed to beat 45 seconds, I would say my design is a good one.

One interesting thing I found out during the creation of the program is that if you try to loop through more than a million numbers in the bitmap the performance goes down drastically. I am not completely sure of the root cause of this but I believe it has to do with the fact that the computer can only hold so much information in its fast memory at a time and if you exceed this them the program has to start using slower forms a memory.