

Calculating Prime Numbers Comparing Java, C, and Cuda

Philip Matuskiewicz (pjm35@buffalo.edu)

December 8, 2009

Available online “CSE710 Project Presentation”:

<http://famousphil.com/schoolarchive/prime710.zip>

(This PowerPoint and Source Code are Zipped)

The Problems to solve

- Calculate prime numbers using both sequential and parallel code
 - Discover an algorithm that works
 - Learn CUDA to implement the algorithm in parallel
- Compare the runtime of several methods of computation
 - Learn about timing methods in C and Cuda
- Provide definitions of common parallel terminology

Parallel Terminology

- OpenMP - Open Multi-Processing – Splits the problem up into processor cores on multi core CPUs
- Node- hardware computer containing a tesla unit, a cpu, memory, etc – Dell Poweredge if I recall
- Tesla Unit – a GPU unit containing 4 graphics cards with 128 processors each
- CPU – Central Processing Unit – The brain of the computer
- GPU – Graphics Processing Unit - similar to the CPU but slimmed down in functionality to handle quick computations in parallel
- MPI – Message Passing Interface – Divides a problem up among several computers in a cluster
- *CUDA - Compute Unified Device Architecture - Very similar to C or Fortran, allows parallelization algorithms to run on GPU processors*
- Speedup – How much faster can a parallel machine solve your problem when compared to a sequential machine
- Thread – a subroutine that is ran in parallel to solve a chunk of the problem. Typically hundreds of threads exist together to break up and solve a problem.
- Memory
 - Shared – Every processor can access this memory – tends to be slower to access
 - Distributed – Each processor has memory that only it can access
- Race condition – When multiple processors attempt to write at the same memory location at the same time
- Credits - Wikipedia

Overall Goals

- Devise a sequential algorithm (computer procedure) for calculating prime numbers
 - Operable Range: 1-1,000,000 (will change later)
 - Obtain the runtime of the algorithm
 - Sequential Languages:
 - Java – High Level with Object Oriented Features (Easy)
 - C – Low Level Procedural Language (Slightly more difficult)
- Output prime numbers / runtime information to:
 - A text file
 - The console [System.out.println(), fprintf()]
 - Verify output with known prime number table
- Parallelize the algorithm
 - Same constraints as sequential algorithm
 - Written in Cuda using an arbitrary number of nodes
 - Compare the runtime to that of a sequential algorithm

The Algorithm

- Take a number n
- Divide (every number from 2 to the squareroot of n) into n
 - If the divisor leaves a remainder, the number is prime.
- I've verified this works up to 1 million proving my algorithm will work using the unix diff command
- Sources:
 - <http://science.irank.org/pages/5482/Prime-Numbers-Finding-prime-numbers.html> - Algorithm Explanation
 - <http://primes.utm.edu/lists/small/millions/> - verification table

Procedure – Pseudo Code

- ***Temp = Ceiling (square root (number));***
- ***From j to Temp, test:***
 - (number \neq j) AND (number mod j == 0) ARE TRUE
 - This is NOT a prime number, break out of the loop now
 - Else this could be a prime number
 - continue the loop until the end
- If the loop finished, echo the number and test the next prime number
- Multiples of 2 are ignored because they are never prime
 - Exception -> 2

Procedure Results in Java

Calculating the primes to 1 million

- See the provided Java source code
 - Compile using `javac Finder.java`
 - Run: `java Finder <number>`
- Execution time was 2630 ms on Magic
- The ci-xeon nodes can't run Java

Procedure Results in C – to 1 million

- Please see the included C source code
 - Compile using: `cc -lm seq.c`
 - Run using: `./a.out <number>`
- Timing code has been fixed- millisecond resolution
- Time Results:
 - **CI-XEON-10:** two Quad Core Xeon (E5430) Processors operating at 2.66GHz
 - 707 milliseconds used by the processor.
 - **CI-XEON-4:** Dual Core Xeon (X5260) Processor operating at 3.33GHz
 - 408 milliseconds used by the processor
 - **Magic:** two Dual Core Xeon Processors (5148LV) operating at 2.33GHz
 - 1140 milliseconds used by the processor

Parallelization Strategy

- Divide the problem and solve it
 - Give all available processors a subset of the problem
 - Devise an algorithm to give processors an equal amount of numbers
 - Use up to all 13 nodes and all available 12,000 processors + to divide the problem to see what kind of speedup there is
 - Maximum number of calculations per test will be 1000
 - $\text{Sqrt}(1,000,000) = 1000$, I'm not exceeding 1 million... yet
 - Most numbers will be weeded out and never make it to 1000
 - mod (2) and (3) are ran first and tend to weed out non primes quickly

Biggest Overall Issues

- Dividing the numbers up
 - My mind just doesn't work in a parallel mindset
 - Surprisingly, this was the sticking point that took me the longest
 - Later slides will describe 2 versions of what I did and the runtime results of both
- I found that <<<1, 512>>> was the configuration that seemed to work best for my problem
 - Allocates 1 grid with 1 huge block with 512 threads.
 - Thread id accessed via threadIdx.x
 - Using this, I could easily calculate the offset and testnumber for each cuda core

Biggest issues con'd

- Magic compiles differently than the ci-xeon-n nodes
 - This got me at least 10 times during my progress in the past few weeks, although I did know better
 - Possibly due to no Tesla unit being connected to Magic
- Race condition
 - I used a shared array during portions of my project and due to mistakes with my division of the problem, I ran into this frequently, although not intended

Expected Results Reminder

- When running the algorithm with CUDA
 - I expect to see what takes 1 second to run on a sequential processor to take about 100ms on the parallel processors
 - This estimate was back in November before I started.

My first working parallel solution

- This was completed the week of Thanksgiving...
 - I figured this would be a good start for improving my code
 - Basically give each kernel invocation 512 numbers to test to match the thread id and add them to a master array at the sequential level.
 - This is a highly sequential code with a lot of extra parallel overhead.
 - Led to excessive looping of the kernel which cost a lot in terms of time!

Initial Findings

- Runtime on Magic with 13 nodes: 583 seconds
 - This a 583x Speed DOWN and HIDEOUS
 - Lots of overhead in creating, copying and working with arrays
- I discovered that it takes roughly 8 seconds to create each MPI process and get the kernel invoked and ready to do something. This makes the above result somewhat believable.
- **There has got to be another solution that is far better!**

My fixed attempt

- After considering everything on paper, I came up with the following formula:
- `int numberpes = (int)ceil(((double)ctmax/2)/((double)numprocs);`
 - This calculates how many odd numbers each processor should calculate
- `Int start = ((2*numberpes)*procnum)+1;`
 - This calculates the starting point for each processor
- Using the above formulas, I was able to have each node figure out where it should have the kernel start calculating and how far the kernel should go...
- I then added the functionality to the code to access more than 1 cuda device per node using similar methods. The code for this improvement exists in the cpe folder of the source code.

How the kernel does calculations

```
int ix = threadIdx.x; //tells me my thread id
//below is how many times do I need to iterate considering 512 processors
int iterations = ceilf((double)numperpes/512);
    for (int x=0; x<iterations; x++){
        int offset = (x*512)+ix;//offset is this
        int testnumber = (start + (2 * ix)) + (512*x*2);
        //this is the number I'm calculating
```

- I pass the kernel an array and I set the array at offset to the testnumber only if it is a prime number.
- For non prime numbers, the array at position offset is set to 0.
- If the test number is 1, it is set to 2 in the kernel
 - (1 isn't prime by definition, 2 is but is an exception, so this is how I handle this)

Runtime of the Efficient Code

- My runtimes will only cover the most efficient code that I have... located in the cpe folder.
- 9 Seconds is the average runtime for 1 million numbers
 - This is about an 9x speed DOWN!
 - OUCH!!!! This goes totally against what I expected so far.

Recall

- From the earlier version of my code... I found that the kernel takes roughly 8 seconds to start up... this is overhead time.
- So my main question to you:
 - **What number should I calculate to before parallel becomes quicker than sequential C?**
 - Before you take a guess:
 - lets look at some results to make an educated guess

Magic's Configuration

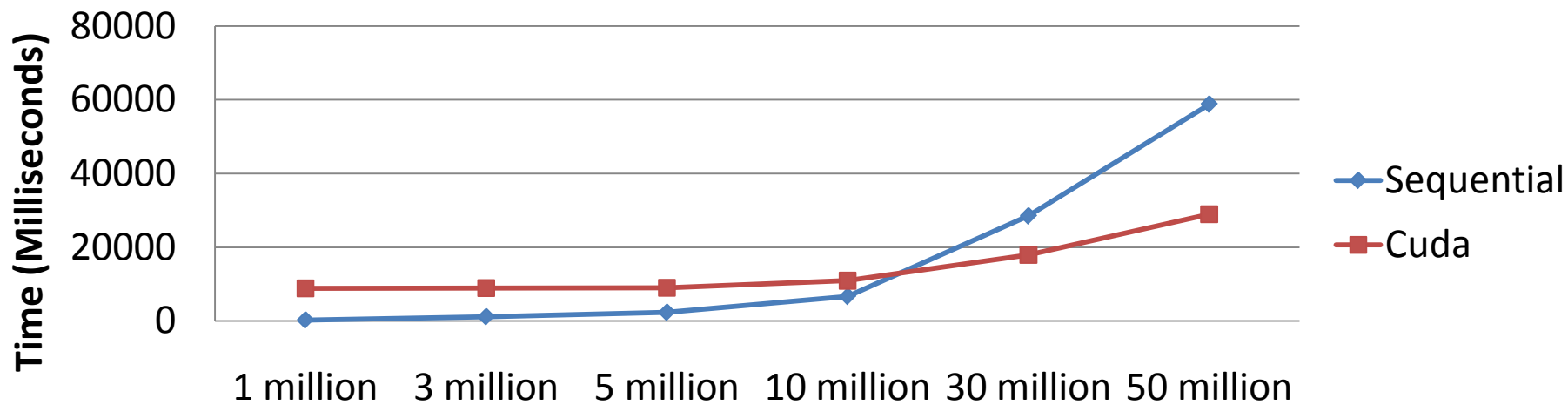
- Ci-xeon-1 through Ci-xeon-8 have 2 CPUs
- Ci-xeon9-Ci through xeon-13 have 8 CPUs
- For my code to run over all 13 nodes effectively, I can only reserve 2 of the 4 GPU cards in the tesla units across 13 nodes.
 - 1 GPU per processor core on a node
- I varied my code a bit, testing different parts of the cluster.

Sample Runtime Results Xeon 1-8

- Reminder: Speed: 3.33Ghz/core – 2 GPGPUs per node

Count To this #	C Sequential Runtime	Cuda Runtime
1 million	256ms	8839ms
3 million	1146ms	8934ms
5 million	2332ms	8989ms
10 million	6658ms	10934ms
30 million	28474ms	17942ms
50 million	58775ms	28967ms

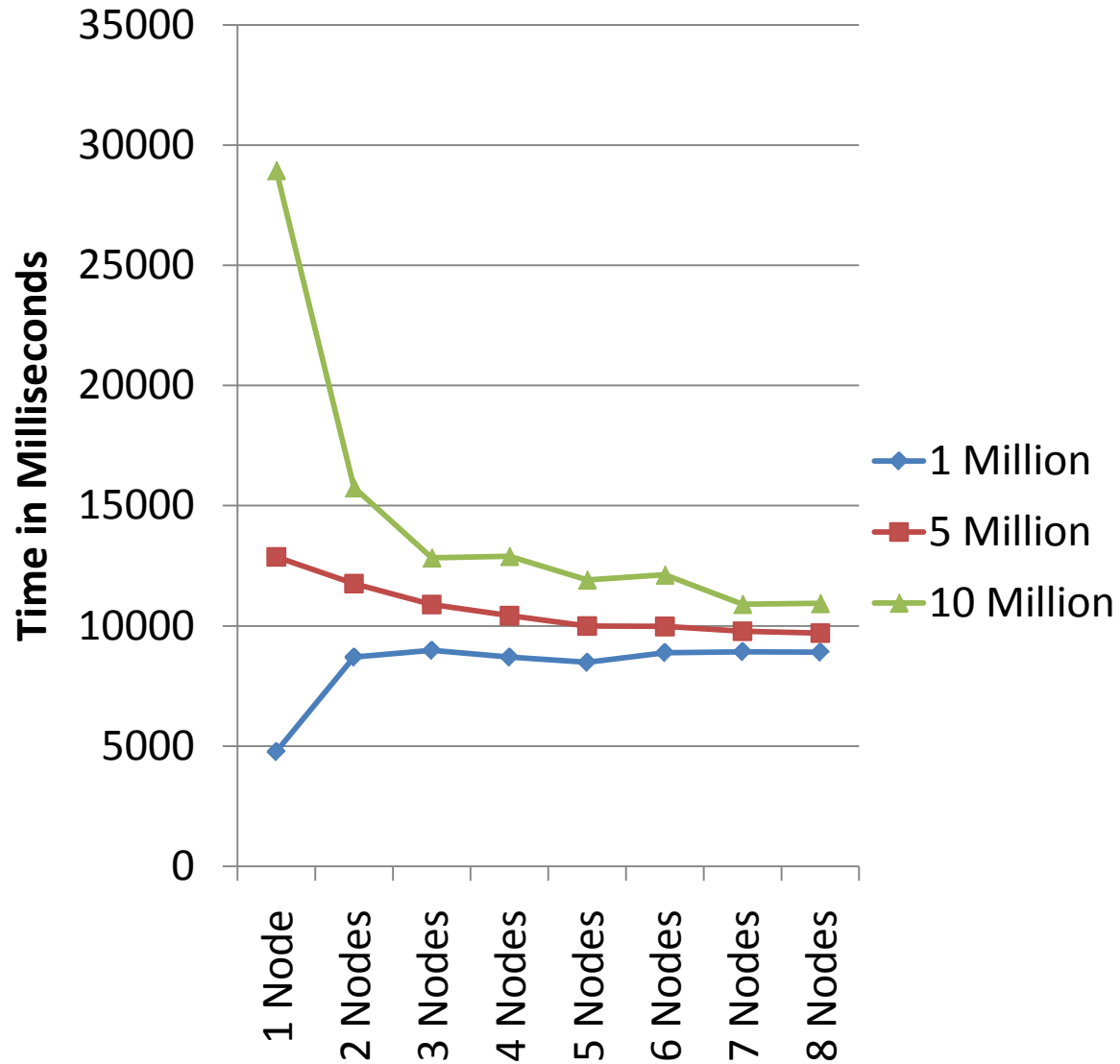
Runtimes



Varying Processors Results Xeon 1-8

- Reminder: Speed: 2.66Ghz/core – 4 GPGPUs per node

Number of Nodes	Count to 1 million	Count to 5 million	Count to 10 million
1	4771ms	12875ms	28946ms
2	8700ms	11763ms	15757ms
3	8981ms	10889ms	12837ms
4	8699ms	10424ms	12900ms
5	8481ms	9992ms	11918ms
6	8888ms	9977ms	12126ms
7	8927ms	9779ms	10906ms
8	8907ms	9701ms	10939ms

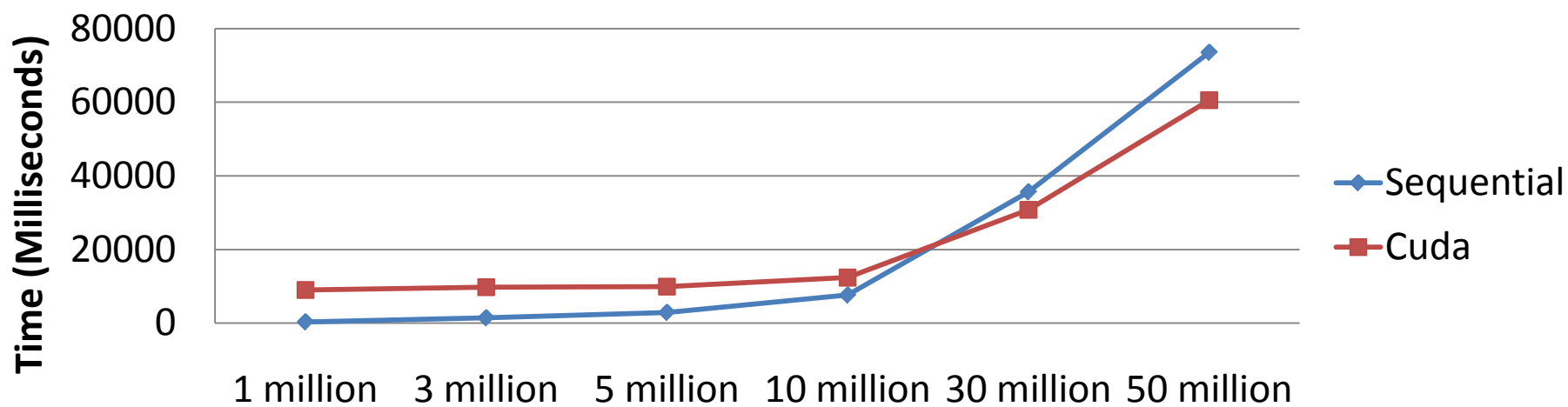


Sample Runtime Results Xeon 9-13

- Reminder: Speed: 2.66Ghz/core – 4 GPGPUs per node

Count To this #	C Sequential Runtime	Cuda Runtime
1 million	327ms	8987ms
3 million	1432ms	9735ms
5 million	2898ms	9882ms
10 million	7602ms	12374ms
30 million	35636ms	30782ms
50 million	73543ms	60587ms

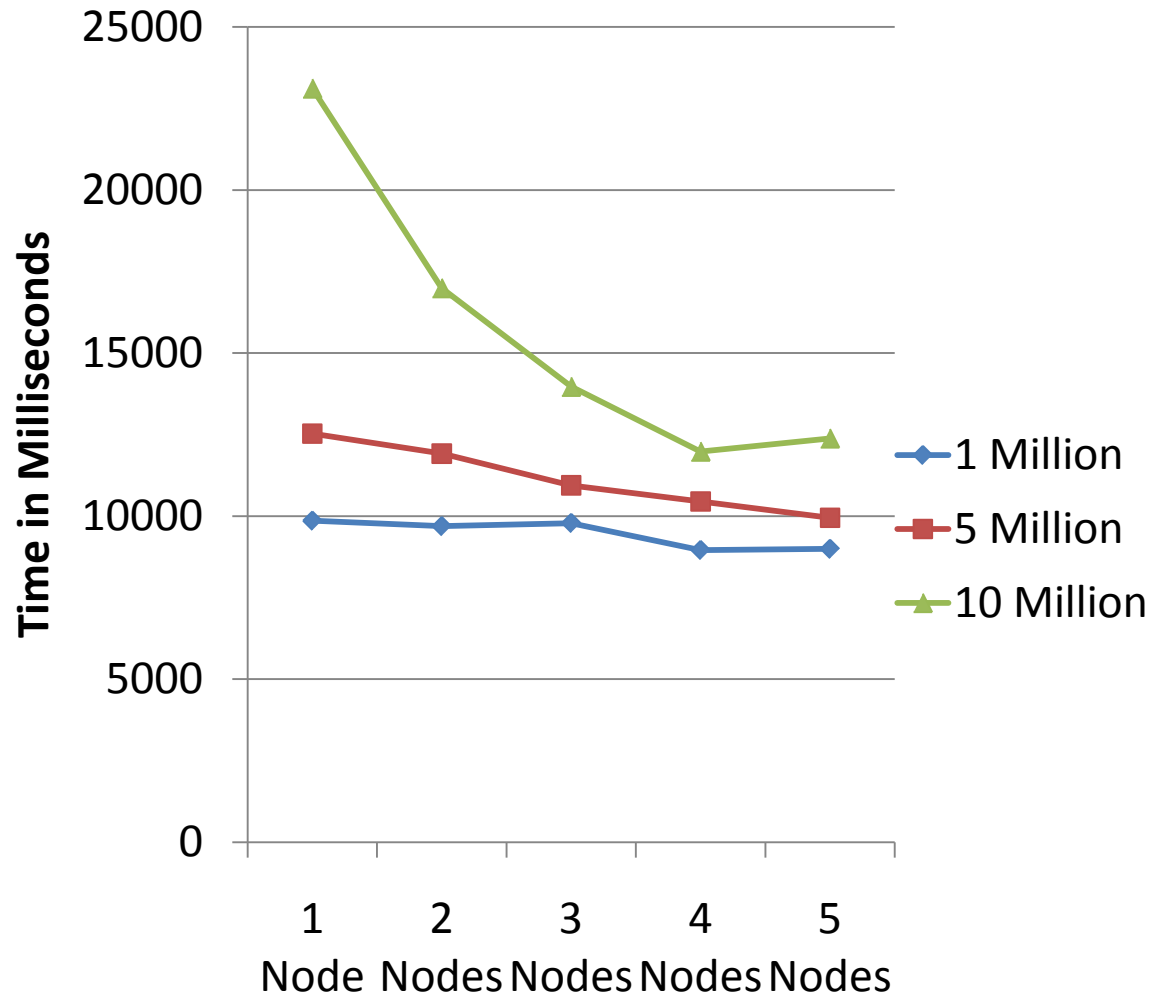
Runtimes



Varying Processors Results Xeon 9-13

- Reminder: Speed: 2.66Ghz/core – 4 GPGPUs per node

Number of Nodes	Count to 1 million	Count to 5 million	Count to 10 million
1	9858ms	12528ms	23118ms
2	9690ms	11918ms	16995ms
3	9787ms	10945ms	13971ms
4	8958ms	10451ms	11982ms
5	8993ms	9949ms	12386ms



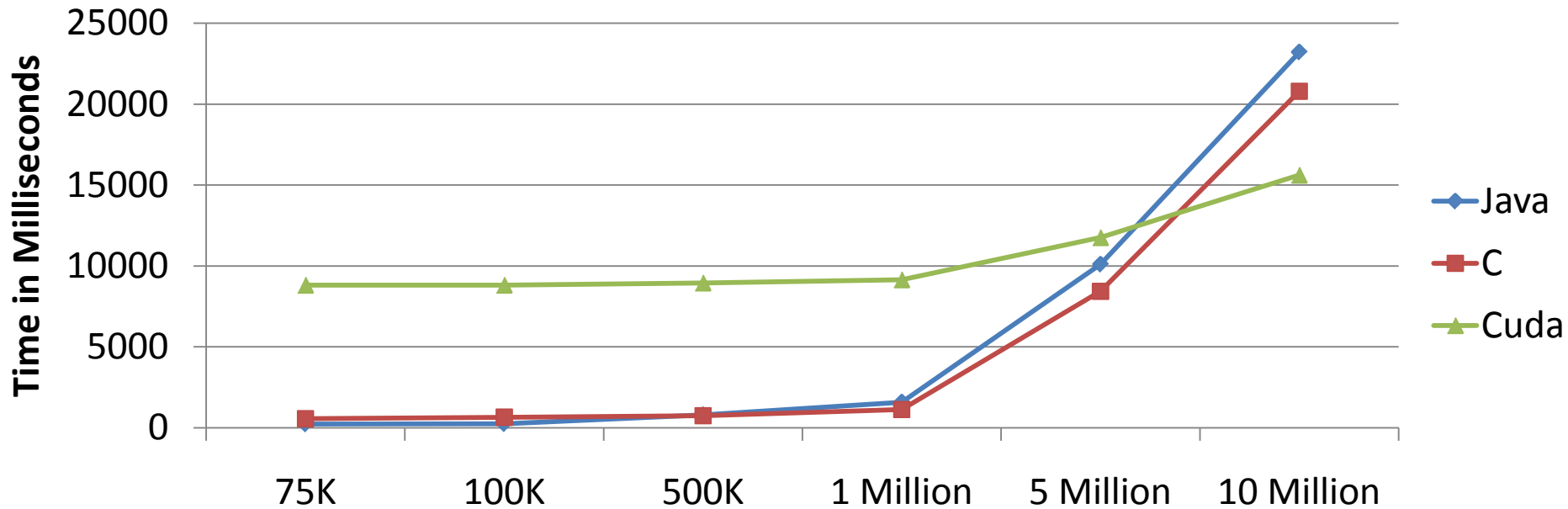
Using all 13 Nodes

- Reminder: When using all 13 nodes, I can only reserve 2 GPUs per node. Therefore, I use 1 GPU per node.
- I'm comparing this to sequential C and Java code on the head node (magic.cse.buffalo.edu).
- This answers my original question to my satisfaction...
- **When does parallel code become advantageous over sequential code when calculating prime numbers?**

Using 13 Nodes with 1 GPU each

The results for c and java sequential code obtained on Magic's Head node running at 2.22Ghz. The cuda code is essentially running at ¼ capacity in this analysis

Count to	Java	C Sequential	Cuda : 13 Nodes / 1 GPU/Node
75,000	234ms	558ms	8820ms
100,000	252ms	655ms	8823ms
500,000	801ms	750ms	8958ms
1 Million	1589ms	1130ms	9154ms
5 Million	10105ms	8440ms	11763ms
10 Million	23213ms	20793ms	15626ms



Other Thoughts

- I found that when I ran the code for 2 GPUs per node, the code became much slower.
 - I believe this is because of some faulty configurations and differences between half the cluster's nodes.
 - It is also possible to contribute this to other users unexpectedly using resources on each node
 - Overall, I believe that Parallel becomes much more efficient once calculating prime numbers far above my original goal of 1 million.
 - For 1 GPU per 13 Nodes, it is about 4 million.
 - To the best of my ability (my actual results were weak)... When using 2 GPUs per node, it becomes 16 million which is expected.

Questions? / Comments?