CS 475 Parallel Computing
Instructor: Matthew Meyn
Student: **Khuong Luu** (**luukh**)

Project 5

1. **What machine did you run the programs on**
- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- Memory: 16GB, 2400Mhz
- Operating System: Ubuntu 16.04
- Kernel: 4.13.0-37-generic
- **Number of Cores: 4**
- Cache line size: 64 bytes
- **Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 18.0.2.199 Build 20180210**

2. Data Tables

In the following 2 tables, the first row is labels of “data size” and the first column is label for “number of threads”

2.1. Results with Non-Vectorized

	1000	5000	10000	100000	1000000	64000000	100000000
1	1.62	13.44	1.98	96.33	122.01	136.73	137
2	0.3	1.3	5.8	16.72	135.32	263.88	264.35
3	0.23	26.89	2.98	102.15	221.39	382.3	384.54
4	0.18	24.88	12.82	17.23	129.85	504.99	507.39

2.2. Results with Vectorized

	1000	5000	10000	100000	1000000	64000000	100000000
1	7.57	3.52	39.05	17.3	101.55	289.67	287.25
2	0.16	0.71	46.97	128.54	347.1	535.56	543.8
3	3.05	1.45	2.95	20.89	190.15	769.03	767.84
4	2.82	0.75	52.04	268.87	327.99	993.33	996.93

3. Graphs

3.1. Non-Vectorized

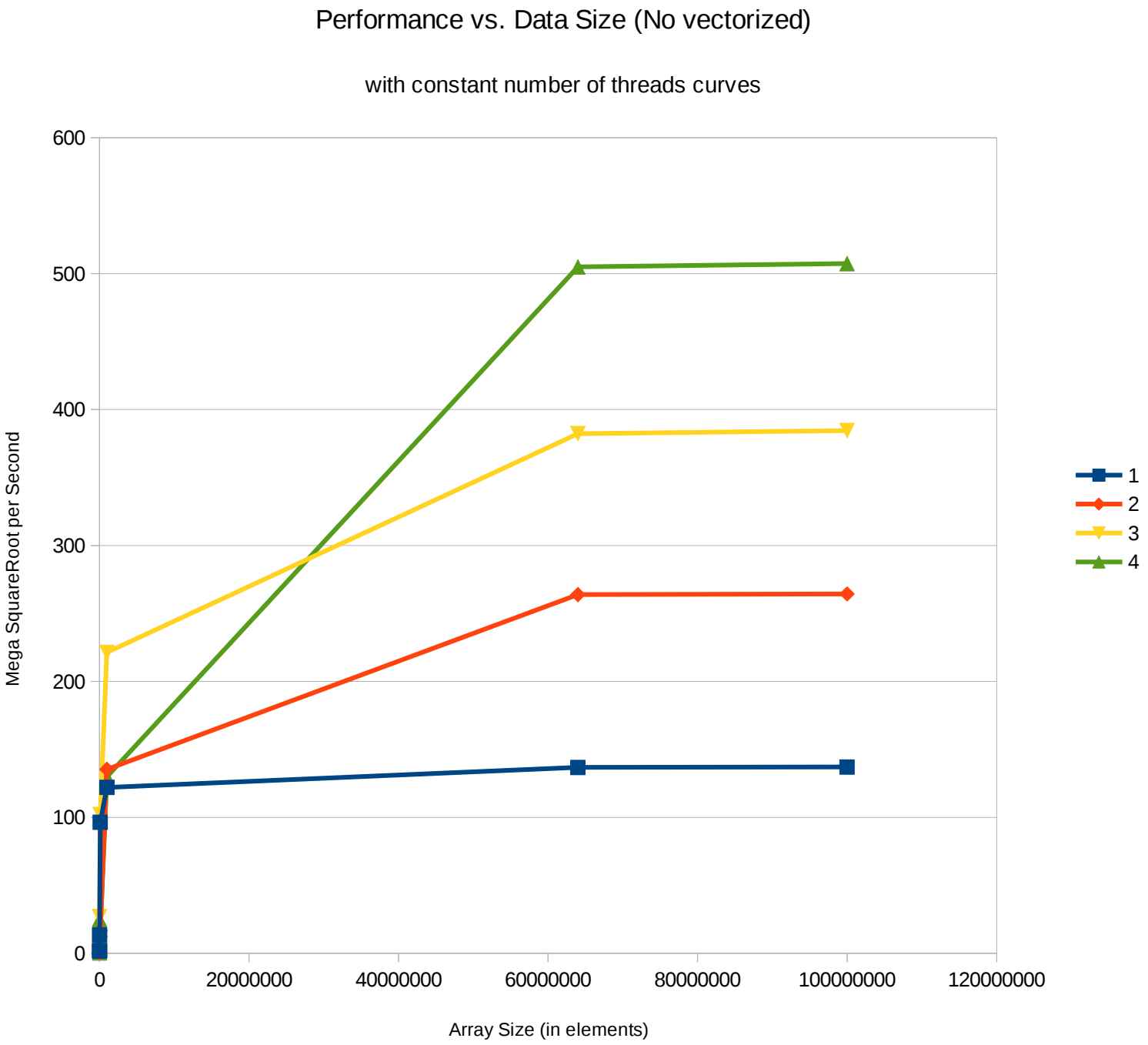

Figure 1a

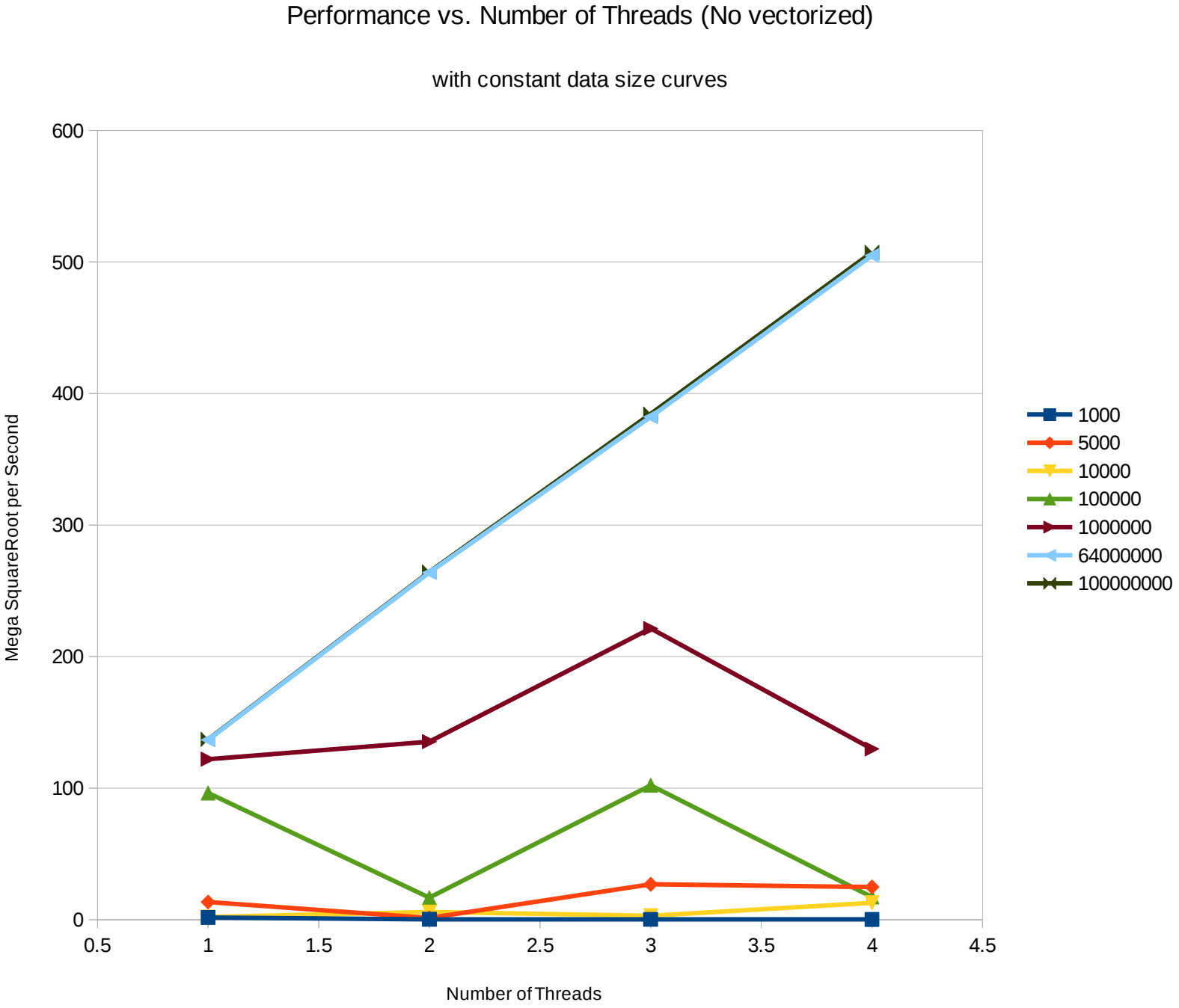
Performance vs. Number of Threads (No vectorized)
with constant data size curves
Mega SquareRoot per Second
Number of Threads
600
500
400
300
200
100
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
1000
5000
10000
100000
1000000
64000000
100000000

Figure 1b

3.2. Vectorized

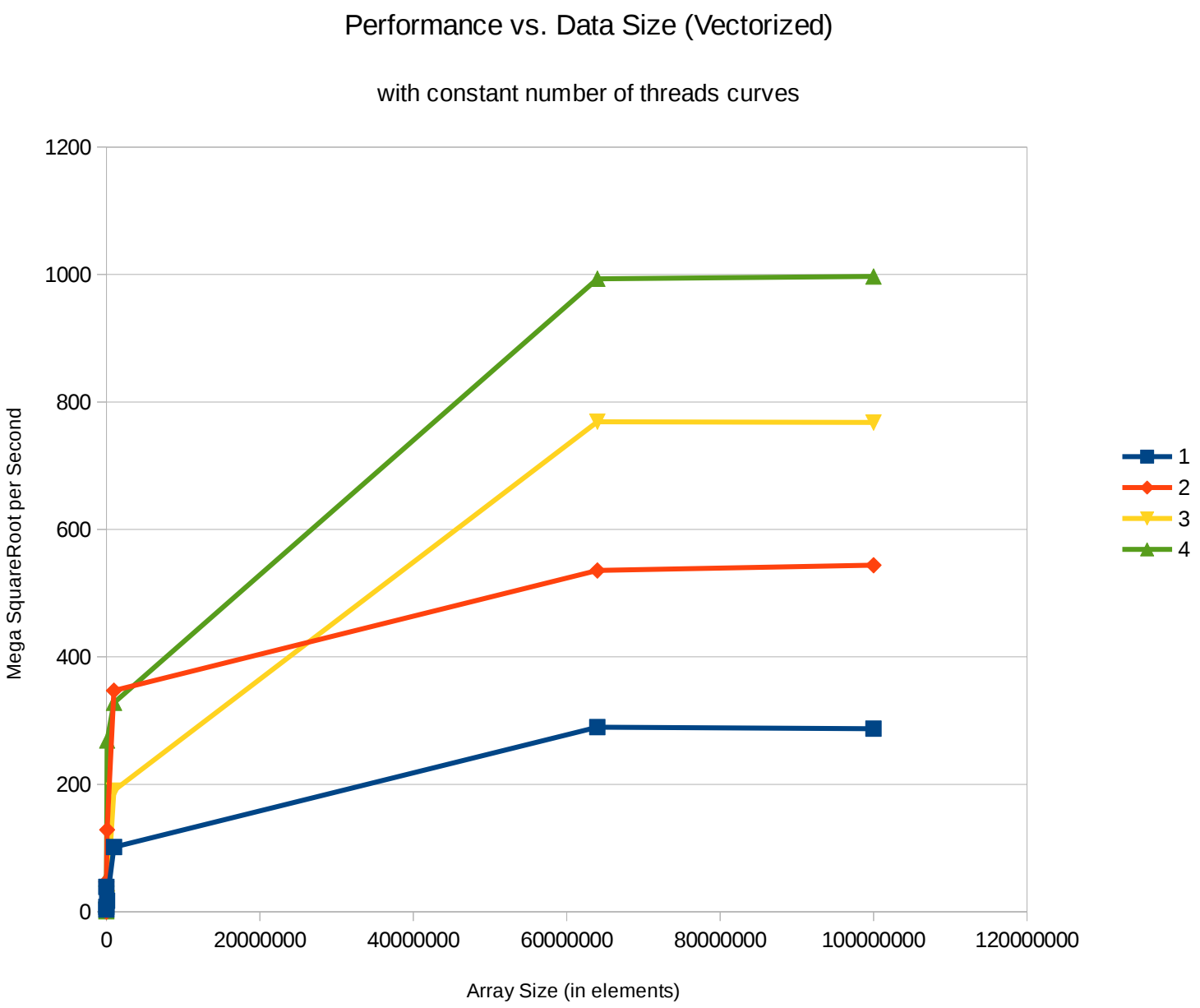

Figure 2a

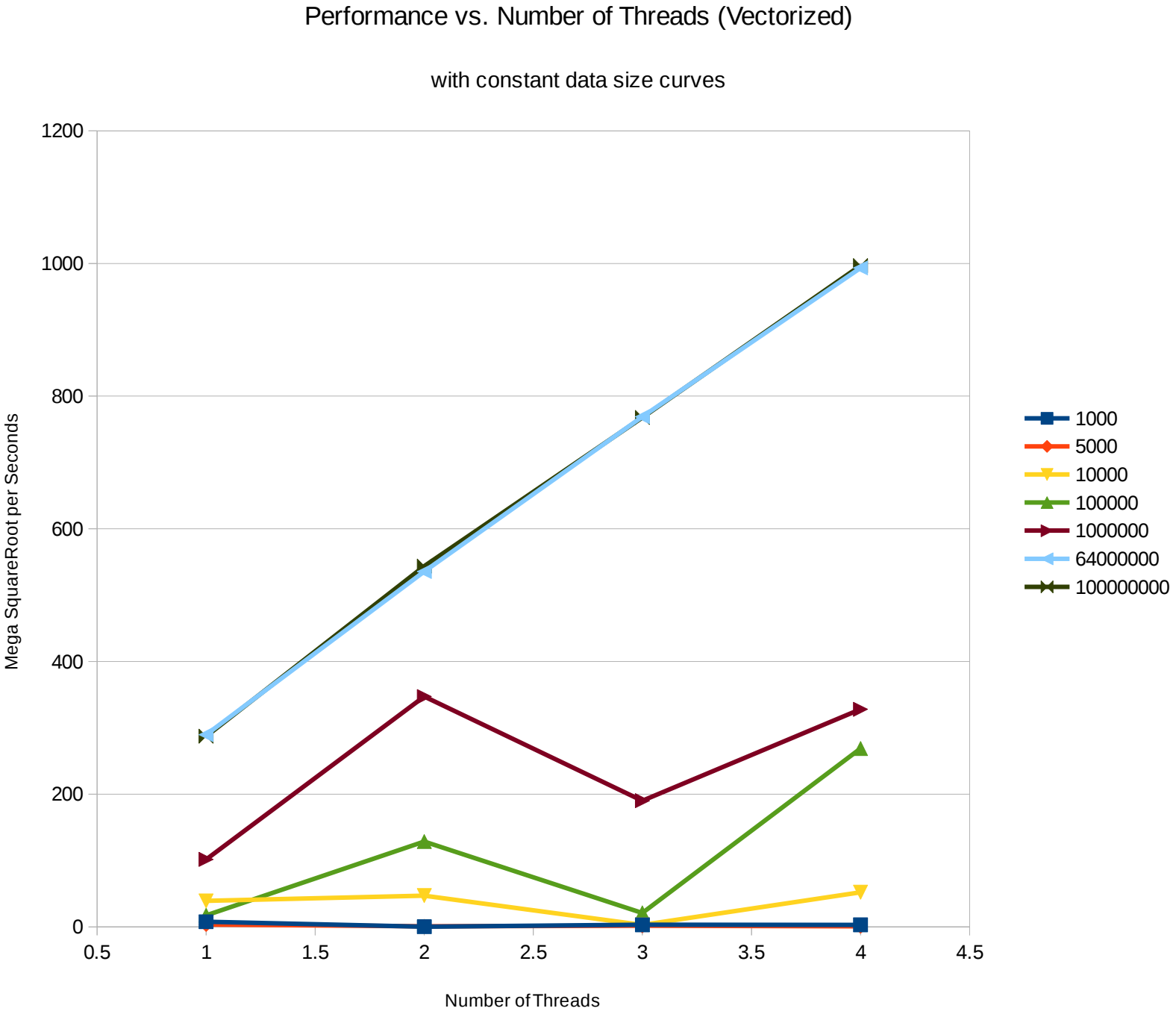
Performance vs. Number of Threads (Vectorized)
with constant data size curves
Mega SquareRoot per Seconds
Number of Threads
0
200
400
600
800
1000
1200
0.5
1
1.5
2
2.5
3
3.5
4
4.5
1000
5000
10000
100000
1000000
64000000
100000000

Figure 2b

4. Patterns and Explanations

4.1. Pattern 1: Vectorized method performs scales better than non-vectorized one

When I look back and forth between Figure 1a and 2a, I honestly didn't notice the difference until I selected and cross-combined 2 pairs of curves together into the same graphs, respectively, below: (Figure 3a and Figure 3b)

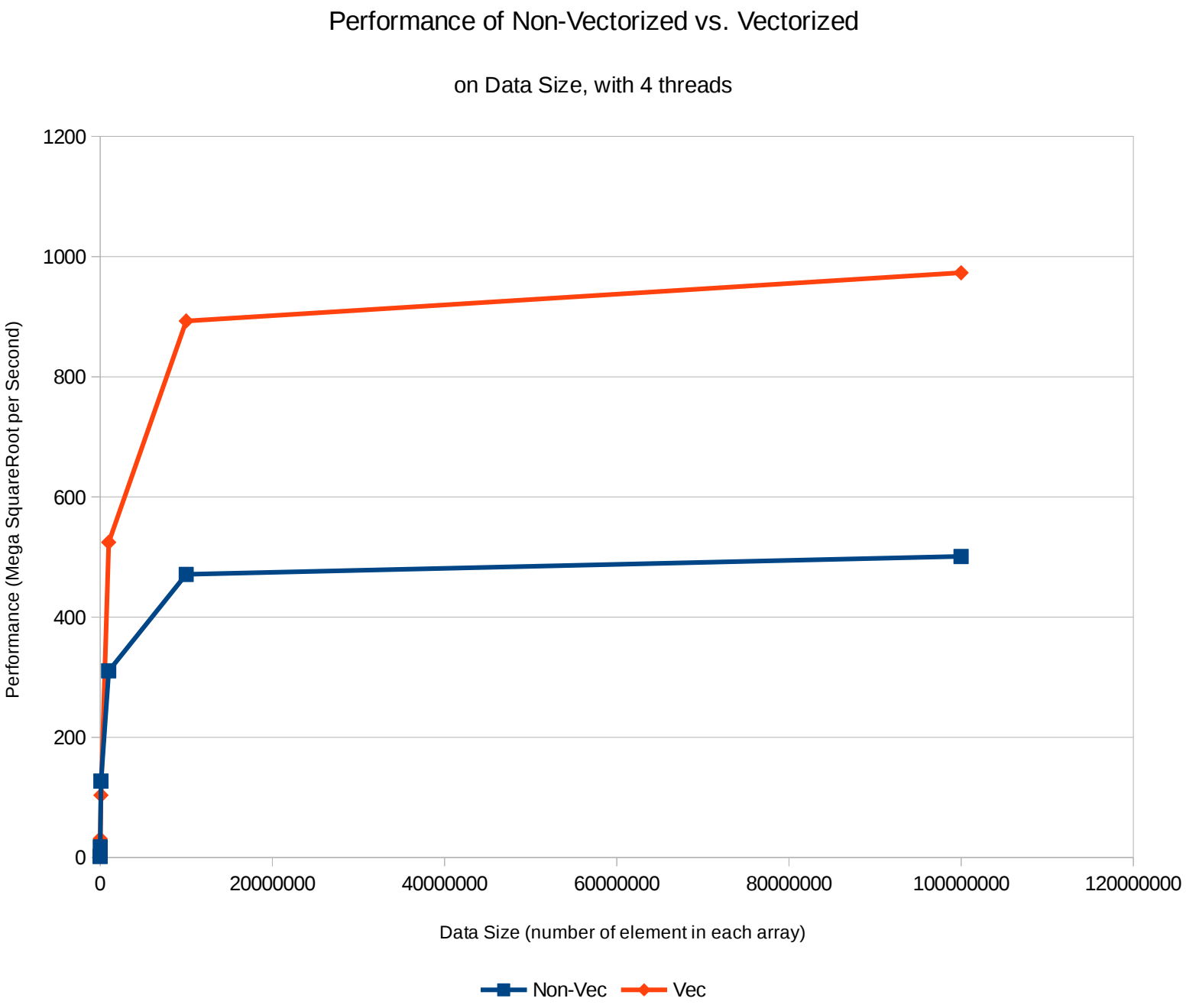

Figure 3a

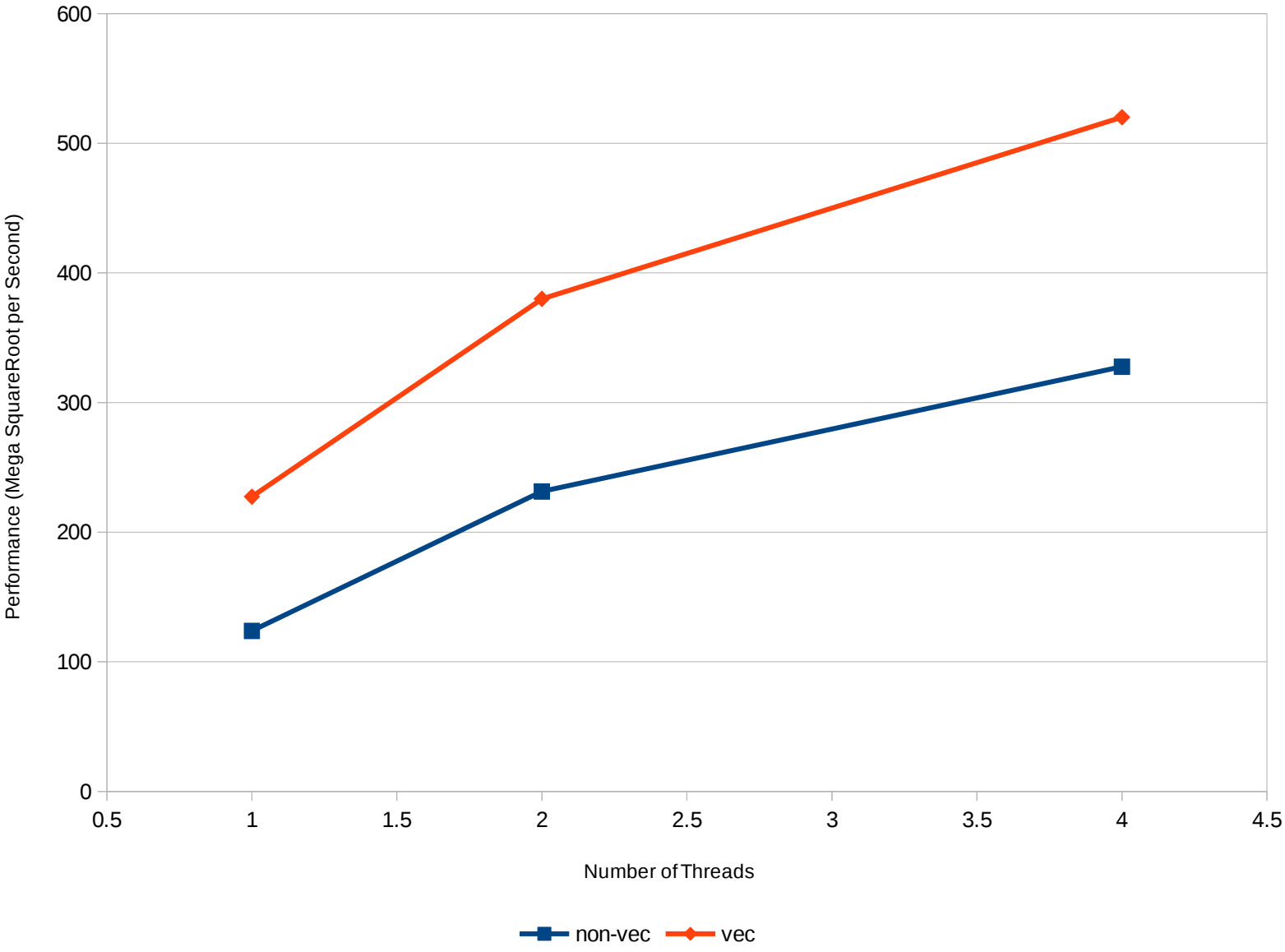
Performance of Non-Vectorized vs. Vectorized
on Number of Threads Used, with Data Size = 1000000
Performance (Mega SquareRoot per Second)
600
500
400
300
200
100
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Number of Threads
non-vec
vec

Figure 3b

It is now clear to see that **the vectorized method performs a lot better** than the non-vectorized method (to more accurate: roughly 2 times better)

That is, with the same data size and number of threads, the performance (in Mega Square Root per Second) of the vectorized program is about 2 times higher than the non-vectorized program.

For the increasing of number of threads, the curve of vectorized method, though already better than its counterpart in value, has the slope notably steeper than the slope of the curve of non-vectorized method.

Explanation:

This is because each thread in the vectorized program performs square root operation on multiple array element at the same time while each thread in the non-vectorized program performs the same square root operation but only on one array element at a time (iteratively). As a result, since each thread in vectorized version perform better, the overall performance of n-threads is better than the non-vectorized version.

Another reason is because in vectorized program, we have another layer of parallel inside each threads (perform multiple operations at once instead of one operation at once) , the upper bound of speed up (compared to 1 threads) of vectorized program is higher than the non-vectorized program, and so the performance of the vectorized program converges later than the non-vectorized program. Take the above case as an example, while the non-vectorized program converges at around 500 Mega Square Root Per Seconds, the vectorized program, though not visualized, may converge at around 2 x 500 = 1000 Mega Square Root Per Seconds (where 2 is how much times the vectorized program performs better than the non-vectorized program), and it converges at this value at a much bigger data size (array size) thus it's the rationale for us to say the vectorized program scales better on data size.

4.2. Pattern 2: Both versions' performance converge at data size of 64 million array elements

For both vectorized and non-vectorized program, data size at 64000000 (64 million elements in array) is the convergence point. Before the data size reaches this value, the more data we put in, the better the 2 programs perform. After the data size reaches this value, the performances stay the same as data size gets larger.

Explanation:

This is because each thread has reach its performance upper bound limit. For non-vectorized program, each thread at its best efficiency can perform 500 MSRPS / 4 threads = 125 MSRPS (MSRPS is "Multi Square Root Per Second). If we vectorize the program, each thread at its best efficiency then can perform 1000 MSRPS / 4 threads = 250 MSRPS (2 times better as we observed and explained above).

But why is the upperbound 64 million and not another number? I don't have a certain explanation for this pattern, but to my best, I think it has something to do with the size of my computer cache line is 64 bytes.

5. What does that mean for the proper use of vectorized parallel computing?

Vectorized parallel computing work best with operation on memories that near each other on cache line. Thus, it is best to align data to the beginning of the cache line. In icpc, I use the -align flag to accomplish this. We can also accomplish this via the “aligned” clause in #pragma omp simd. More specific, my current Intel CPU has 64 bytes cache line and 32 bytes vector size, so the best alignment is 32 bytes, which is two vectors per one cache line.

Also, as each thread performs vectorized operation on <simdlength> element at once, where <simdlength> is the number of element the compiler can vectorize the operation to perform on. Thus, it’d be better to have each threads a large enough number of bytes to perform vectorized operations, at least 32 bytes in my computer.