Vector Scaling On Single and Multiple Threaded Versions

Sam Nowak

1 Introduction

We are testing to see how using multiple threads will affect the speed it take compute

$$Z = c * V + Y \tag{1}$$

where C is a constant and Z, V, and Y are vectors of integers of 1GB size. The first test is just using a single thread to get the base speed. The second test is having threads work on adjacent indexes and then computing their next space after n places, where n is the number of threads. The final test is when we have each thread work on a contiguous block of the thread. We test theses up to 2c threads and observe how they speed up or slow down, where c is the number of cores on the lambda machine.

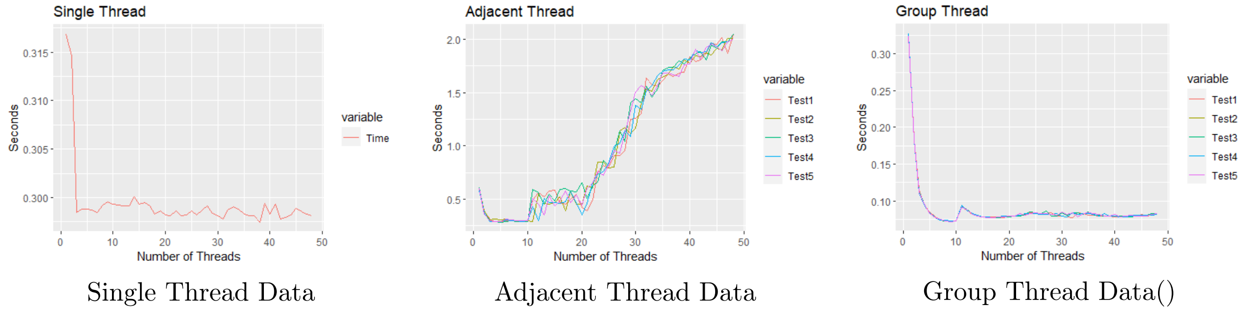
For my hypothesis I predict that both adjacent threads and the blocks threads will be faster than a single thread working on the vectors. As well as them being around the same amount time, but the adjacent threads being sightly faster than the blocks. Over time I predict that they will slow down after going over the lambda core count because then threads are going to fight each other. So, a graph will look like a hump, but not as drastic when coming back down.

2 Method

Using lambda, intellij, and R studio I ran and graphed my data for all three thread versions. Firstly I found out the lambda machine has 24 cores, so I will be running the adjacent threads versions and the block version from 1 to 48 threads to see how different numbers of threads affect the speed. I also found out my vector size by getting the size of 1gb in bytes and dividing by 4 bytes since int in java are 4 bytes. Vectors will have a length of 268435456. Secondly coded everything in intellij using small vectors to make sure my code is running properly. For the adjacent threads I created a class that strides through the vectors by the amount of threads we are using at the time. For the continuous block I divide the vector by the number of threads then made the threads only iterate that many times and the starting position were the other thread would end. Then make sure to use the Stopwatch class to record the time of only when the threads were working and printing that data in the console. After that I put it in the lambda machine and collected my data. Lastly I put my data into a csv file for R studio and graph different relationships to see the different version Vs the single thread model. Also how the adjacent and block version are different from each other.

3 Result

For the single Thread version I ran it 48 times to get an accurate time and over the course it had an average around 0.28 seconds. The adjacent and block thread versions where run from 1 to 48 threads 5 different types to make sure we have constant data.



Single Thread Data Adjacent Thread Data Group Thread Data()

3.1 Adjacent Vs Single thread:

The only speed up I found over the Single thread was from threads 3-10 and then the adjacent threads were always slower than just having one thread. Getting even slower from intervals 10-24 and 24-48. During threads 24-48 the adjacent model had increasing slow down for each thread.

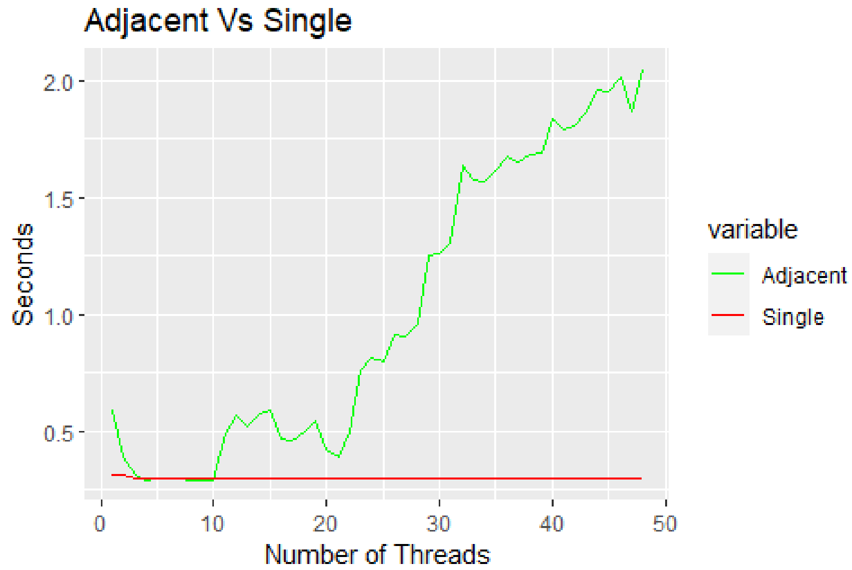


Figure 1: Adjacent Vs Single Data

3.2 Group Vs Single Thread:

The speedup increases rapidly at the start. They both started with the same run-time for one thread. However, after 5 threads speed up from over 0.3 to 0.08 and stayed around 0.08 for the rest of the 48 runs.

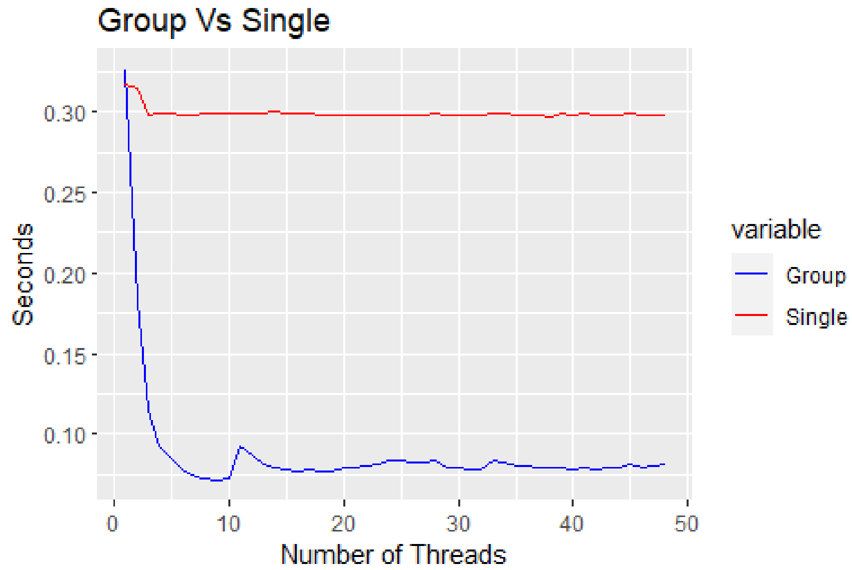


Figure 2: Group Vs Single Data.

3.3 Group Vs Adjacent Thread:

There is a clear difference between the two multi-threaded versions. For about 10 threads they look about the same, but the adjacent threads version is shifted upwards taking more time per run. After 10 threads the Group thread version remains the same speed. The adjacent thread model jump to in time from 10-24 threads and then after 24 threads it starts to slow down even more for each thread added.

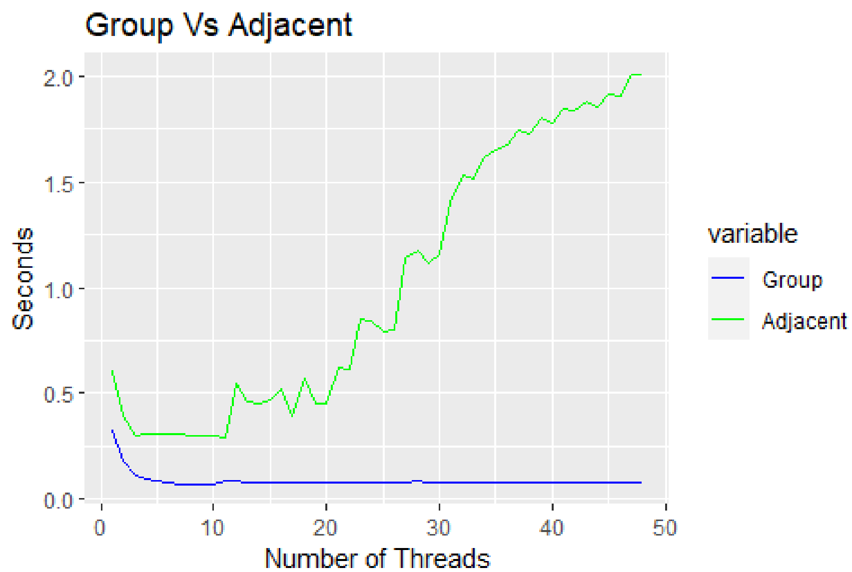


Figure 3: Group Vs Adjacent Data.

3.4 Running One Thread

The singles threaded version and the group version ran the around the same time for one thread. The adjacent version however started around twice as slow as the other thread versions for one thread, which is

very confusing. I felt like they should all be the same time since the code would have identical values at 1 thread.

4 Discussion

For the adjacent threads I found that after going over the core count (24) the adjacent versions suffered slow down for adding more and more threads. I think this is due to having multiple threads on each core, so they must share resources and scheduling.

I was surprised that the group version stayed around the same time for all 48 cores. I think this is from having the threads doing there own sections and being done so they don't have to transverse the entire vectors. So there is more cores free for the incoming threads to finish up.

My data destroyed my hypotheses. I thought the adjacent version would be the fastest, however it was the slowest one and slower than the single thread versions. Also thought that both group and adjacent versions would suffer from slow downs after 24 cores, but only the adjacent versions did. Lastly none of my graphs looked like a hump, I was surprised that the group thread model stayed consistent from 5-48 threads.

5 Conclusion

Depending on how you use threads it can either help the performance of the program or slow it down. As well as the number of threads used can clog up resources and cause it to become slower if you use too many. In this homework having the threads be continuous blocks instead of adjacent threads help the performance of the program.