Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017

Finding Micro-Inversions in Parallel

Background: Last semester I created a project to find micro-inversions in genomes. A micro-inversion is a rare, naturally occurring mutation in DNA where the correct nucleotides exist in the correct location in the sequence, but as the reverse compliment. For example, if there were an inversion of size 4 beginning at position 1 (0 indexing) in AGGTCT, the original would be AGACCT (GGTC -> GACC). While inversions can be of many different sizes, micro-inversions are generally classified as being between 15 to 100 nucleotide bases long while reads of a DNA sequence are generally 100 nucleotide bases long, so those are the lengths that my project assumes. The algorithm that I made last semester has two steps. First, there is a preprocessing pickle step where my program goes through each 100-length nucleotide window in a given genome to calculate its ACGT character density, which is used as a key for a dictionary, which leads to a list of all the positions in the genome sequence with that character density (sequencesDic[numA, numC, numG, numT] -> [List of positions in genome with the given character density]). The second step is to take in reads and simulate every inversion at every position in the read until a micro-inversion is found by using the proposed character density to reference the dictionary created in the preprocessing pickle step and comparing it to the genome locations in the character density dictionary list value. It is important to note that detecting micro-inversions is still a topic that is being researched in the computational genomics community, so I was satisfied with finding a bruteforce-like method of finding micro-inversions since it worked. However, Professor Langmead commented that it seemed my project was "ridiculously parallelizable", and so we attempted to create a parallel version of the code (Projectcode/PastMethod/parallelInversions.py). The attempt did not work because of raze errors and was quickly scrapped.

Thesis: The startup costs of spawning multiple processes to parallelize this project can be avoided provided that the genome sequence is long enough. On a relatively small genome sequence, the time complexity of finding a single micro-inversion is O(ir^2) , where i = max inversion length and r = the read length. However, if the genome is of notable size, the time complexity is changed to O(iGr^2), where G is the size of the genome. The larger G become, the more positions need to be checked at a given character density. Unlike max inversion length and the read length, the genome size is likely to change from run to run and it can become very large (up to ~250million nucleotides per chromosome) so it is the most important factor in the time complexity formula to consider when parallelizing the program.

Steps Taken: When starting the project, the first task I gave myself was refamiliarizing myself with the code. In doing so, I realized that the preprocessing file calculated the character density inefficiently, and in fixing that inefficiency, I sped up the preprocess step by a factor of 5. I thought fixing a similar issue in the findInversions file would cause a healthy speedup as well. However, as can be seen in Graph 1 below, on larger genome sizes, the speedup was completely erased. That result only reaffirmed my belief in the thesis that true speedups will only be achieved if they consider the size of the genome, and fixing character density calculations does not do that.

I pointed my attention to the area of the code where the length of the genome comes into play (see line 39-41 in findMicroInversions.py -- Essentially, the larger the genome, the more string comparisons are necessary since there will be more positions in the genome with the same character density). I decided to use python multiprocessing's pool and map in findMicroInversionsCompareMultiProc.py to parallelize the string comparisons by sending each process its own position in the list of possible positions to compare with the proposed micro-inversion, returning true if there's a match to end the cycle. However, this version of the program performed very poorly when compared to the serial version because of the startup costs of creating new threads each time the program reaches the string comparison step. It performed so badly that I didn't even run it on 1000 reads like I did with every other test because it would have taken 8 days to complete for the largest genome test I had. I thought part of the problem with its performance was related with how many different items were mapped to processes in pool.map, so I created findMicroInversionsCompareGroupedMultiProc.py which grouped the genome list positions into n groups to lower the startup costs of sending a list position to a process, where n is the number of processes specified. The division happened in such a way to ensure all processes would do at most one more string comparison than any other, thus ensuring there wouldn't be any significant skew between processes while avoiding some of the startup costs associated with the non-grouped version. While the compareGrouped version did perform significantly better than the non-grouped version, it was still much slower than the serial version. I realized that meant that each process was not doing enough work, so I started looking at ways to give each process more work.

I decided the best way to give each process more work was to loop unroll. The string comparisons occur within a while loop that adjusts the position of the proposed inversion within a read, so I thought why not give each process the chance to handle several iterations of the whole loop. But in doing that thinking, I realized that there was no reason to not let each process simply handle all of the inversion starting positions. That would work since the positions while loop is within a while loop for inversion length. However, if I could do that for the positions while loop, I could also do it for the inversions size loop since the method to find an inversion within a read is called inside of a loop for all the reads. So, I decided to make my next attempt for parallelization to be one that parallelizes all of the reads. Unfortunately, the parallel version of the code that was attempted in the original project tried to do a similar task and it did not work, so I tried figuring out what caused its error. I was able to pinpoint two issues – one, the failed parallel method made use of global variables that were shared across all processes that should not have been shared, and two, the lack of communication between processes caused the outputs to print over each other. To solve the first issue, I passed each variable as a

parameter to the various methods instead of relying on global variables, and to solve the second issue, I used python multiprocessing's Queue to store all of the outputs before printing them once every read had been processed. These changes worked, so I again created a grouped and a non-grouped version similar to how I did for the string compare version. Both performed much better than the serial version, but the non-grouped version performed slightly better than the grouped version. I believe the switch in which performed better between grouped/non-group happened because in the parallel read version, each process had enough work to do on its own to negate the startup costs of having each process take in a new read each time, and so the initial startup cost to organize the reads into groups is the difference in timing between the two methods.

While I was content with multi reads' performance across all of my tests in terms of total time, the average time to handle a single read went up because of the startup costs associated with each read. Being content with those results would be ignoring the scientific problem associated with micro-inversions. In reality, there would not be 1000 reads for a program to find micro-inversions in because micro-inversions are very rare in nature. A program like bowtie2 (made by Professor Langmead!) would first align the regular reads, and only the small amount of reads that weren't aligned would be inputted into a program like this one. So, speeding up the total number of reads processed shouldn't be my goal, but rather, I need to speed up each individual read's processing time. So I decided to not unroll the inversions loop like I did in the multi read version, and instead let each process handle one inversion size at a time. Unlike the string compare parallel version, this method of parallelization is at risk of having more than one proposed inversion being found at a time, so I decided to use multiprocessing's Queue once again. In this method, it's used such that once an inversion is found in the read, its information is added to the queue, thus incrementing the total size of the queue to at least 1. Each process first checks to see if the queue's size is greater than 0, and if it is, it ends. Then, the queue's content is printed out thus returning the total count back to 0. (If in the rare case two or more insertions happen for the queue at the same time, all extra entries are also cleared). While I did not attempt to make a grouped version of the parallel inversions program because of the results I saw with the groupedReads version, I did make an "inverseInversion" method that's very similar to the regular parallel inversion method. In the 7 other versions of the code I made, possible inversions are tried on a read in descending order because larger inversion sizes need to be tried at fewer positions and smaller inversions have to be tried in several places. For example, a length 99 inversion only needs to be tried at positions 0 and 1 in a 100 length read while a length 20 inversion would need to be tried at positions 0, 1, 2…78, and 79. However, in the inverseInversion method, I switched the order to be ascending instead because then each process would have more work to do, which I thought would help mitigate startup costs further. However, even though it performed much better than the serial version, it performed considerably worse than the regular parallel inversion method.

For all eight of the methods I created for this project (two serial, two parallel string comparison, two parallel reads, and two parallel inversions), I ran tests on the same 500k nucleotide long sequence, 5million sequence, and 50million sequence. On the six parallel versions, I set the number of processes to 8. Due to the lengthy amount of testing needed for this project, every test was run on Ugrad-21 instead of on AWS. Other than the comparison parallel methods, each test was run with the same 1000 reads generated by inversionGenerator.py. For the two comparison parallel methods, I only ran around 50 reads on each because of the long time necessary for each read. After those initial tests, I ran tests varying the number of processes used in the two different parallel inversion methods on the 50million dataset. Finally, I ran additional process-varying tests for the non-inverse parallel inversion method on the 500k and 5million datasets, as well as a new 200million dataset that I created for the purpose of illustrating the effect of the genome size on the results of the potential parallelism.

Findings: Overall, I found that that findMicroInversionsLengthMultiProc.py is the best version of the program because its average time for finding a microinversion is lowest. (See Graph 2) While the two multi read methods are faster overall, they ignore the key property of microinversions: they're too rare to warrant wanting overall execution speed to be lower instead of average time per inversion. (See Graphs 3a and 3b) While I can artificially create 1000 reads for the program to read, in reality, reads would first be aligned together by a program like bowtie2 and then the few unaligned reads left over would be fed into my program to search for inversions in the data.

Interestingly, the serial versions of the program performed better than the parallel inversion version on the smallest dataset, 500k. However, that result is to be expected because each process would have less work to do when working with a smaller sequence since there would be fewer string comparisons to do at each inversion length / starting position; so the result is in agreement with my thesis of genome size affecting the effectiveness of parallelism.

Initially, the bad results from the two compare methods surprised me. I was expecting its performance to improve relative to the serial method as the genome size grew, but as can be seen in Graphs 4a, 4b, and 4c, that did not happen. However, I realized that I was approaching the problem of giving each process enough work to offset startup costs incorrectly. So, the results from the two compare methods do not go against the idea in the thesis of larger genome sizes helping parallelism because a larger genome size would not affect the amount of work each process would need to do.

Before making the optimal performing parallel inversion method, I first made an inverse parallel inversion method. In that method, instead of going from inversions of size 100 to 15, it went from 15 to 100. I thought the inverse method would perform the same if not better than the regular parallel inversion method because each process would have more work to do, thus mitigating more of the startup cost. However, as seen in Graph 5 and Graph 2, the parallel inversion method performed better than the inverse method at each genome size. The reason for the worse performance can be seen in Graph 5: while inversions of size 15 to about 40 performed well in the inverse method, it does not mirror the good performance of 60-100 in the regular parallel inversion method. Additionally,

the inversions of size 50 and above perform very poorly in the inverse parallel version. The reason for these findings is because when going from high to low, very few comparisons are necessary to traverse down the inversion size (100 only has one check, 99 has two, 98 has 3, ect), but if it starts low and then go high, it requires the program to check many different positions for a single inversion size (15 has 85 checks, 16 has 84, ect).

I also decided to test how changing the number of processes would affect the results so I could get a better understanding of the potential speedup possible as the genome size increased. I had done all my experiments with 8 processes, but as Graph 6 shows, there is no speedup beyond four processes. I was under the impression that the ugrad systens have 8 cores each, but upon further research, I found that four of those cores are virtual cores instead of real ones. (See Image 1 for lscpu output. Very similar to https://unix.stackexchange.com/questions/218074/how-to-know-number-of-cores-of-a-system-in-linux). So, since there are only 4 cores, the average time for a read to be processed increased each time another process was added beyond 4. I ran the number of processes tests on the 500k, 5mil, 50mil, and a new 200m genome sequences to illustrate that as the genome sequence increased, the potential for speedup also increases. (See Graphs 7a, 7b, and 7c)

Amdahl's Law 500k:

$$\frac{109.821334839}{106.706171036} = \frac{1}{(1-P)+\frac{P}{4}}$$

P = 0.037820991189

Amdahl's Law 5Mil:

$$\frac{673.140872}{288.397570133} = \frac{1}{(1-P)+\frac{P}{4}}$$

P = 0.762086

Amdahl's Law 50Mil:

$$\frac{7023}{2266.34916282} = \frac{1}{(1-P)+\frac{P}{4}}$$

P = 0.90306152873

Amdahl's Law 200Mil:

$$\frac{16751.866302}{5173.17929196} = \frac{1}{(1-P)+\frac{P}{4}}$$

P = 0.9215838444

So as can be seen, as the genome size increases, the percentage of the program that is parallelizable increases as the genome size increases. The non-parallelizable parts consist mainly of the startup costs of creating the threads and opening the pickle files containing the genome and the character density dictionary.

Summary: Overall, the results I obtained through the various methods and tests confirm the thesis of my project that an increase in genome size offsets the startup costs associated with spawning multiple processes. The property of parallelism explored, start up costs, affected each of the parallel methods. In the parallel-compare methods, the startup costs of spawning the processes far exceeded any benefit in parallelization. The parallel-reads methods did indicate good speedup relative to their startup costs, but because it ignored the practical scientific problem, it was not explored further. The parallel-inversion methods both showed good speedup relative to their startup costs, and the results on the different size genomes show that the potential for speedup and the speedup observed increases with the size of the genome.

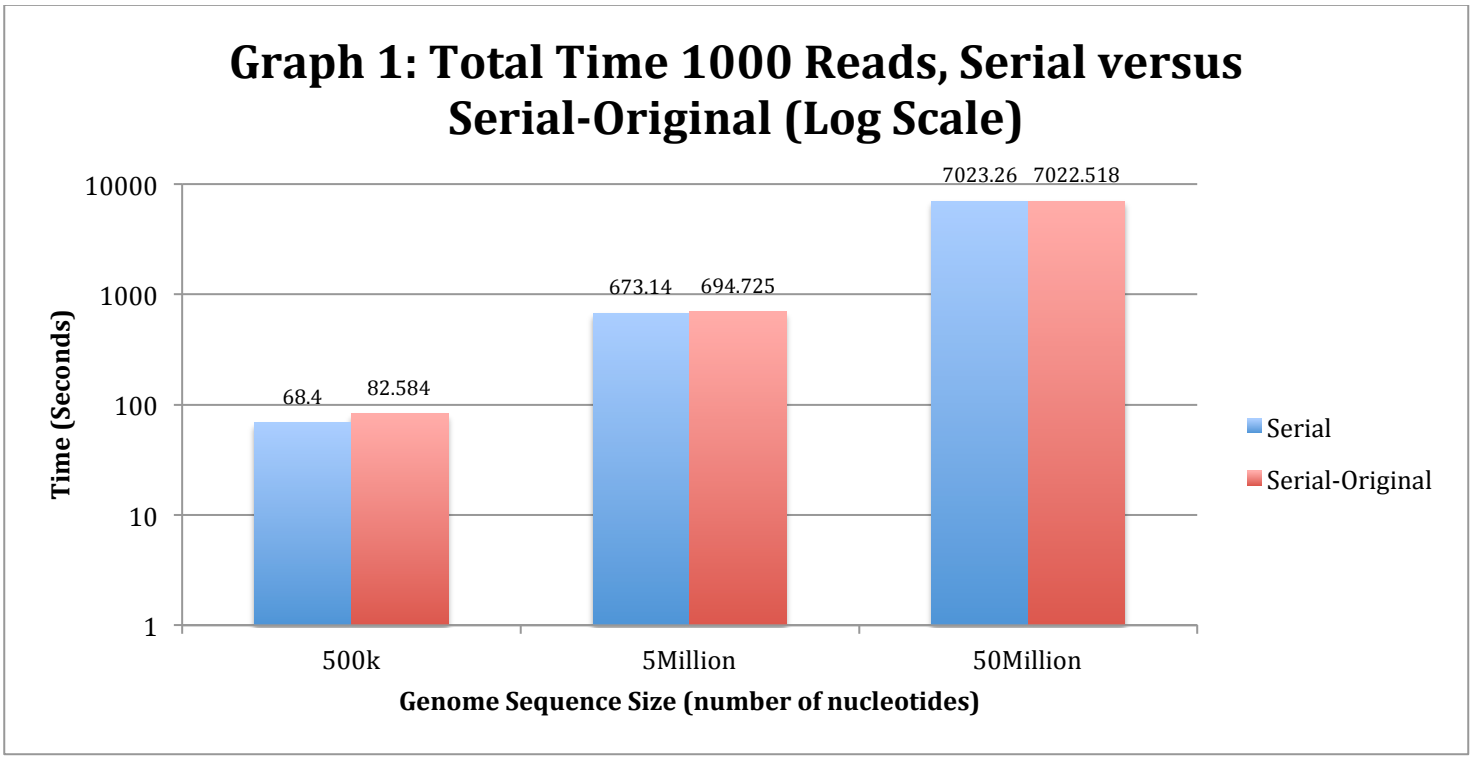
Looking Forward: This topic has become one that I want to continue working on even though this project has come to a close. For most DNA mutations, scoring matrices exist to evaluate how closely related two sequences are. However, because of the difficulty in detecting micro-inversions, one does not yet exist for them. The end goal for me is to use my program as the basis of detecting more micro-inversions to eventually develop a scoring matrix by utilizing mitochondrial DNA comparisons to determine two species relatedness whenever a micro-inversion is found. However, before I can start on that larger journey, I need to improve on my current program further. In terms of parallelism, I need to run further tests to figure out at what sequence size it is more efficient to run parallel code instead of the serial version since Graph 8 shows that the serial version of the code runs faster than the parallel version at smaller genome sizes. Since the genome string is loaded into the program anyway, it should be simple to check the size of the genome

and then decide what version of the code should be run. Additionally, I want to explore if Map-Reduce can be used as a method of parallelism for this project where the character densities are used as keys. In terms of serial improvements, changing the strings in the program to byte arrays would speed up the code because of all the string manipulation that occurs. Major speedups will also occur if I rewrite the program in another language like C++, and my options for parallelism will also increase since I wouldn't need to deal with GIL. In terms of overall program structure, I have been starting the process of rewriting the program with a tree structure instead of a character density - dictionary structure. The preprocess step would involve creating a tree with all the different 100 character combinations + locations, and then the actual program would create a proposed microinversion in an unaligned read and run it through the tree to see if there's a match. The downside with this new method is that the tree would take up quite a lot of space and that there wouldn't be any way of parallelizing the non-preprocessing step aside from parallelizing the reads, which is not necessary for micro-inversions.

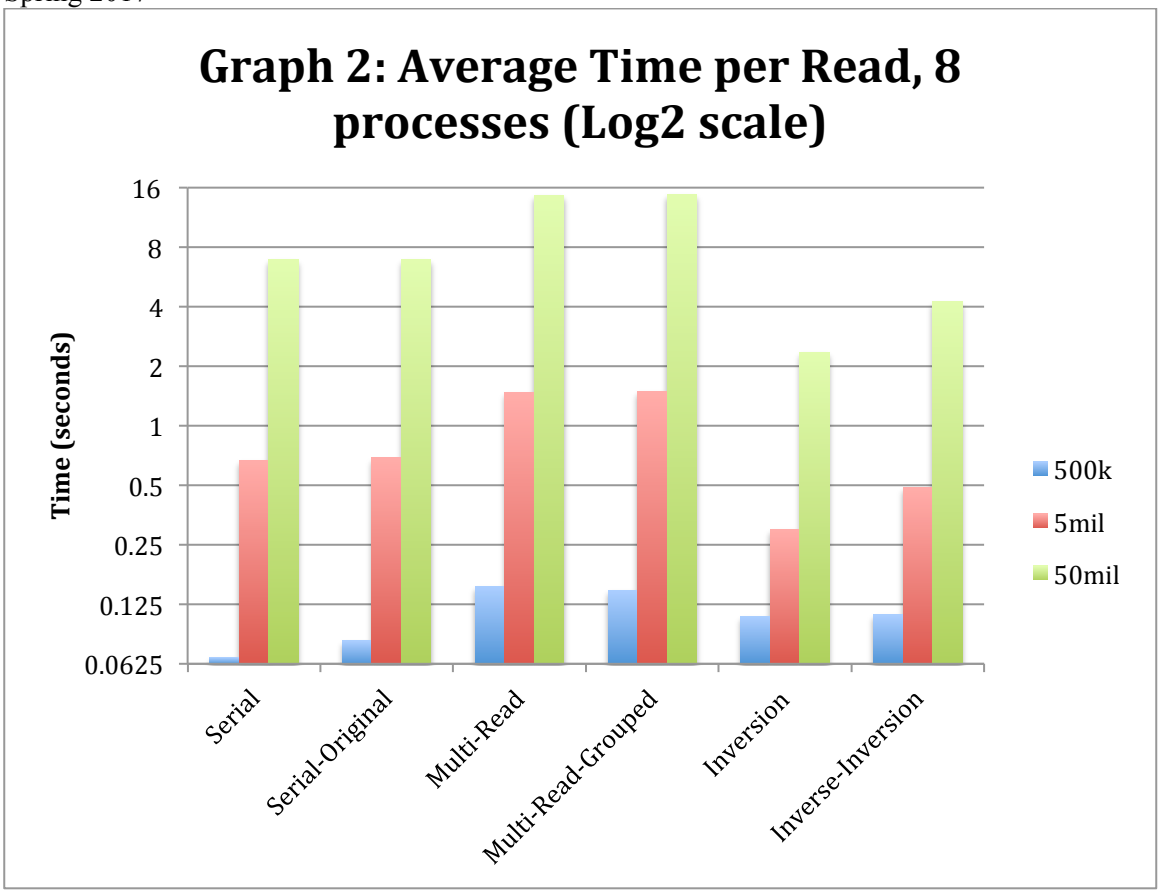
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017
IMAGES AND GRAPHS: All data and graphs can be viewed in GraphsParallelFinal.xlsx

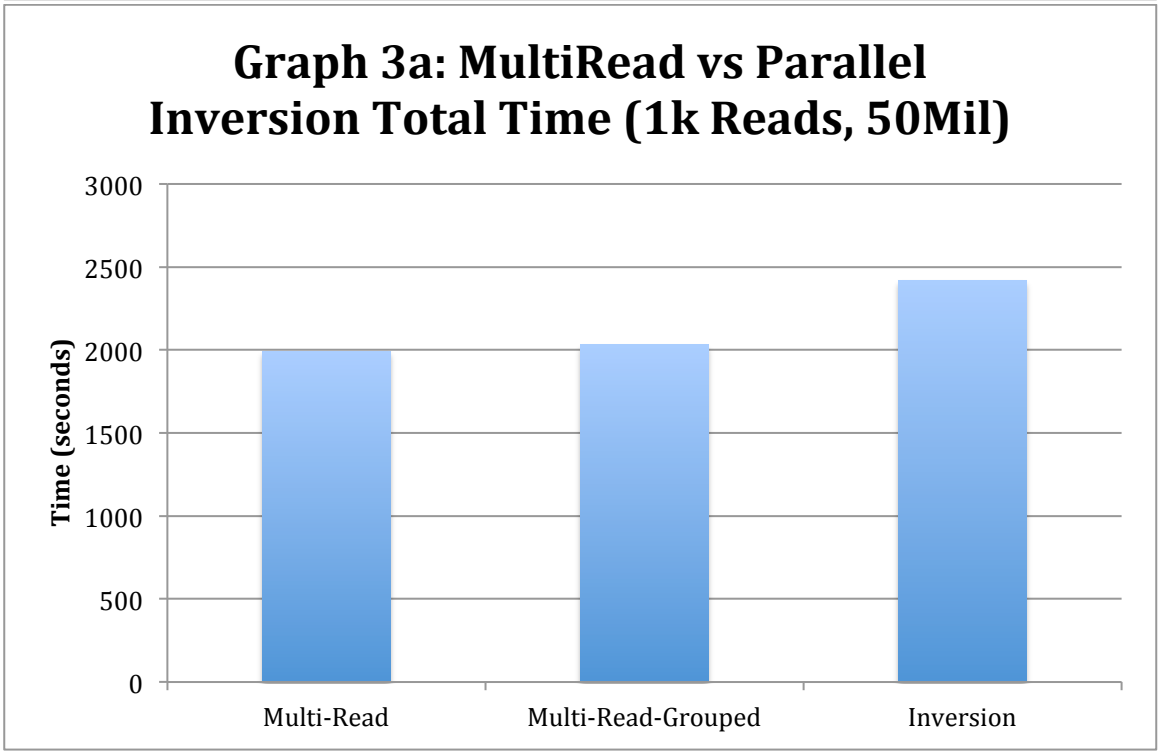
Image One: Results of lscpu on ugrad21

```
[jbajaj1@ugrad21 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Stepping:              3
CPU MHz:               1693.151
CPU max MHz:           3900.0000
CPU min MHz:           800.0000
BogoMIPS:              6784.24
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1g
b rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_t
sc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_tim
er aes xsave avx f16c rdrand lahf_lm abm epb tpr_shadow vnmi flexpriority ept vp
id fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat
 pln pts
[jbajaj1@ugrad21 ~]$
```

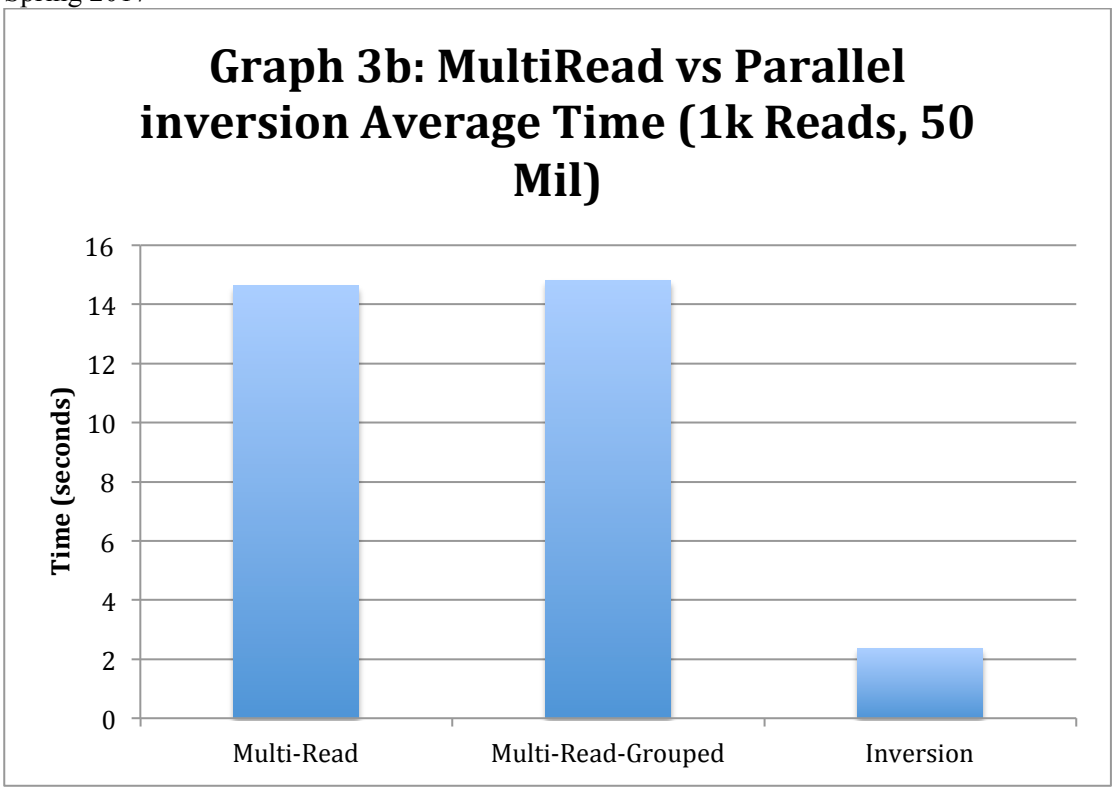


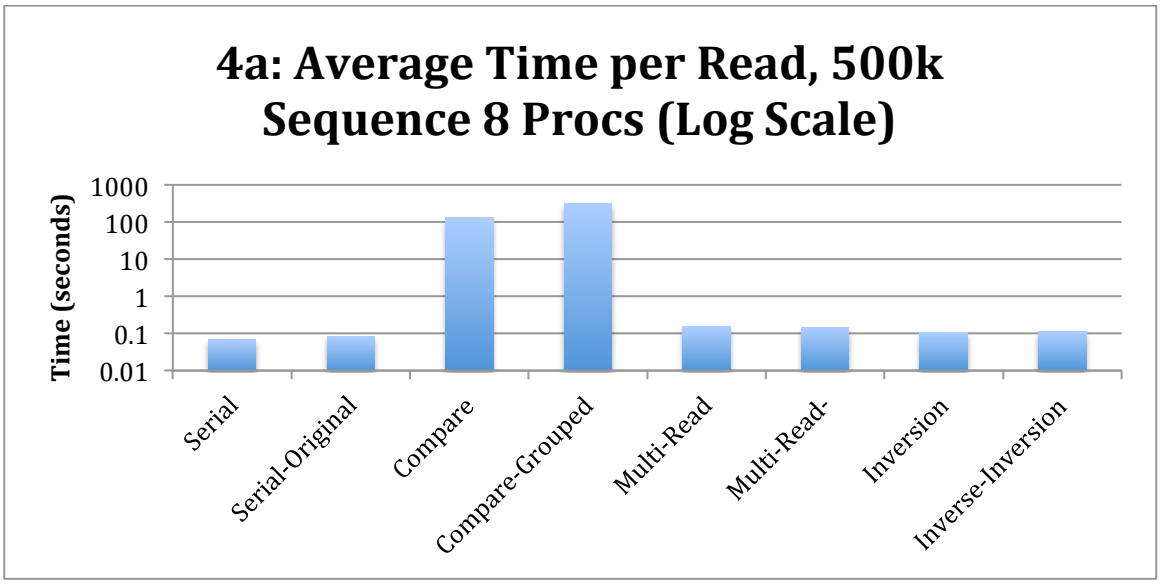
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017



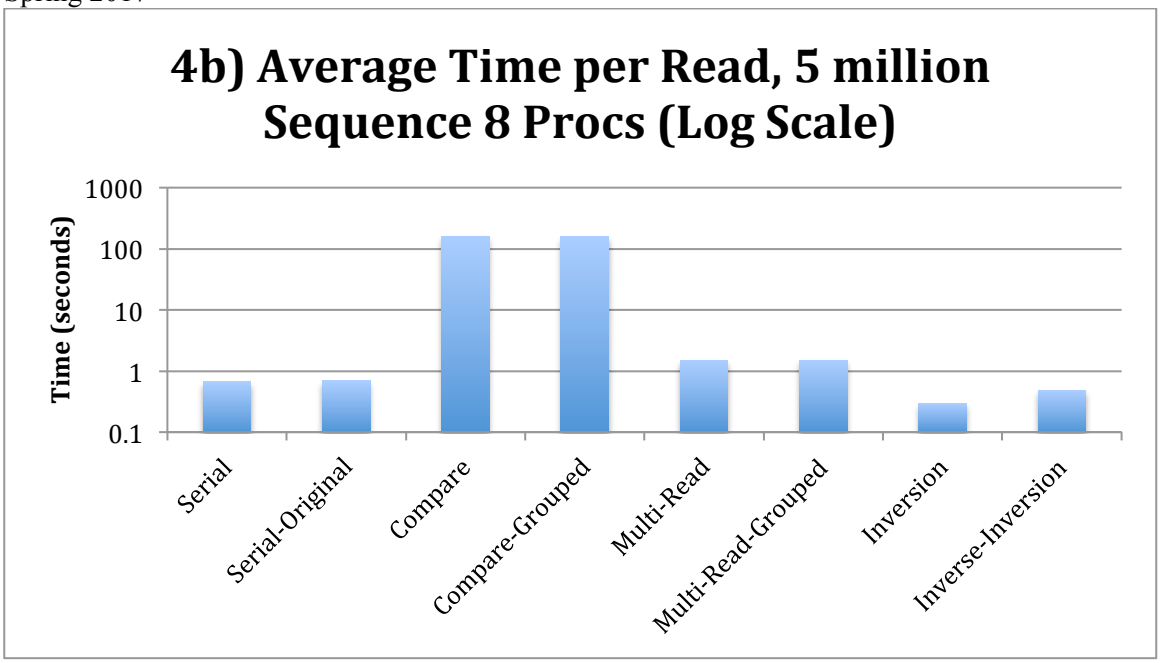


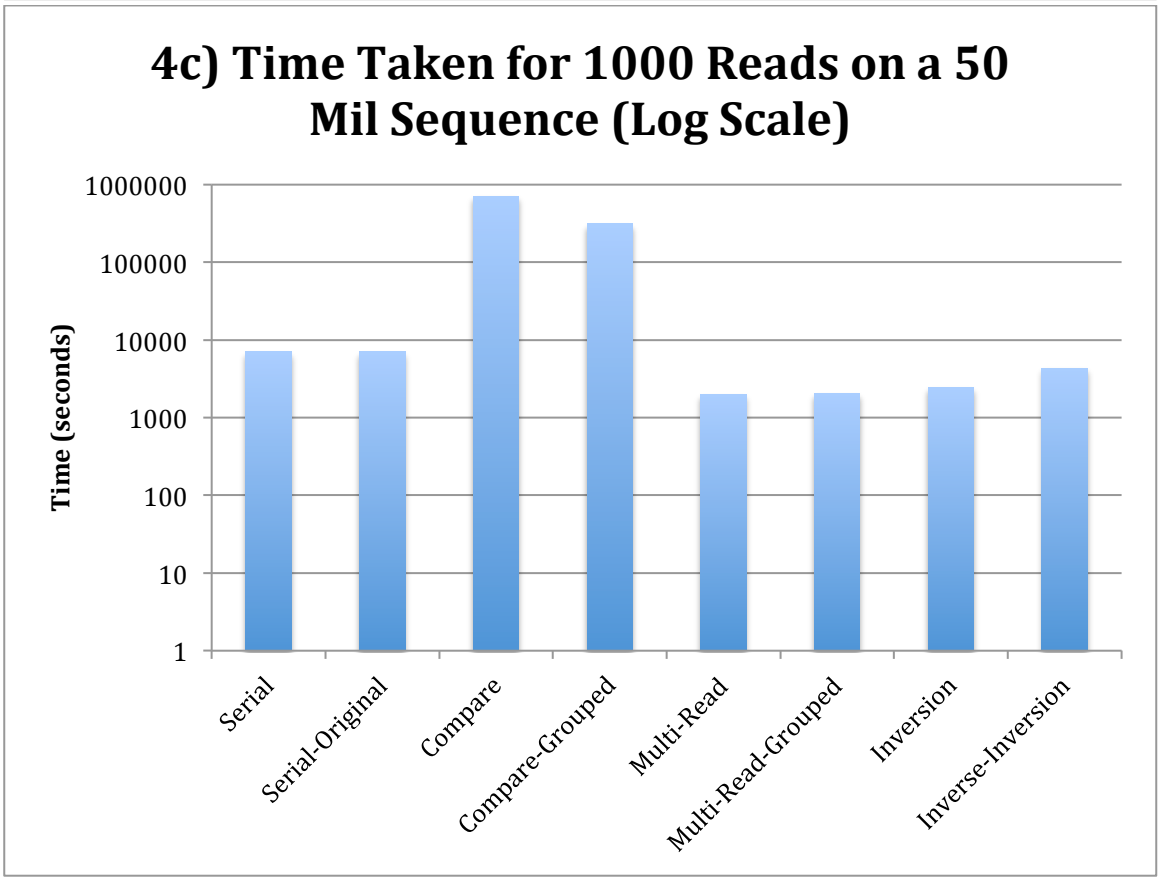
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017





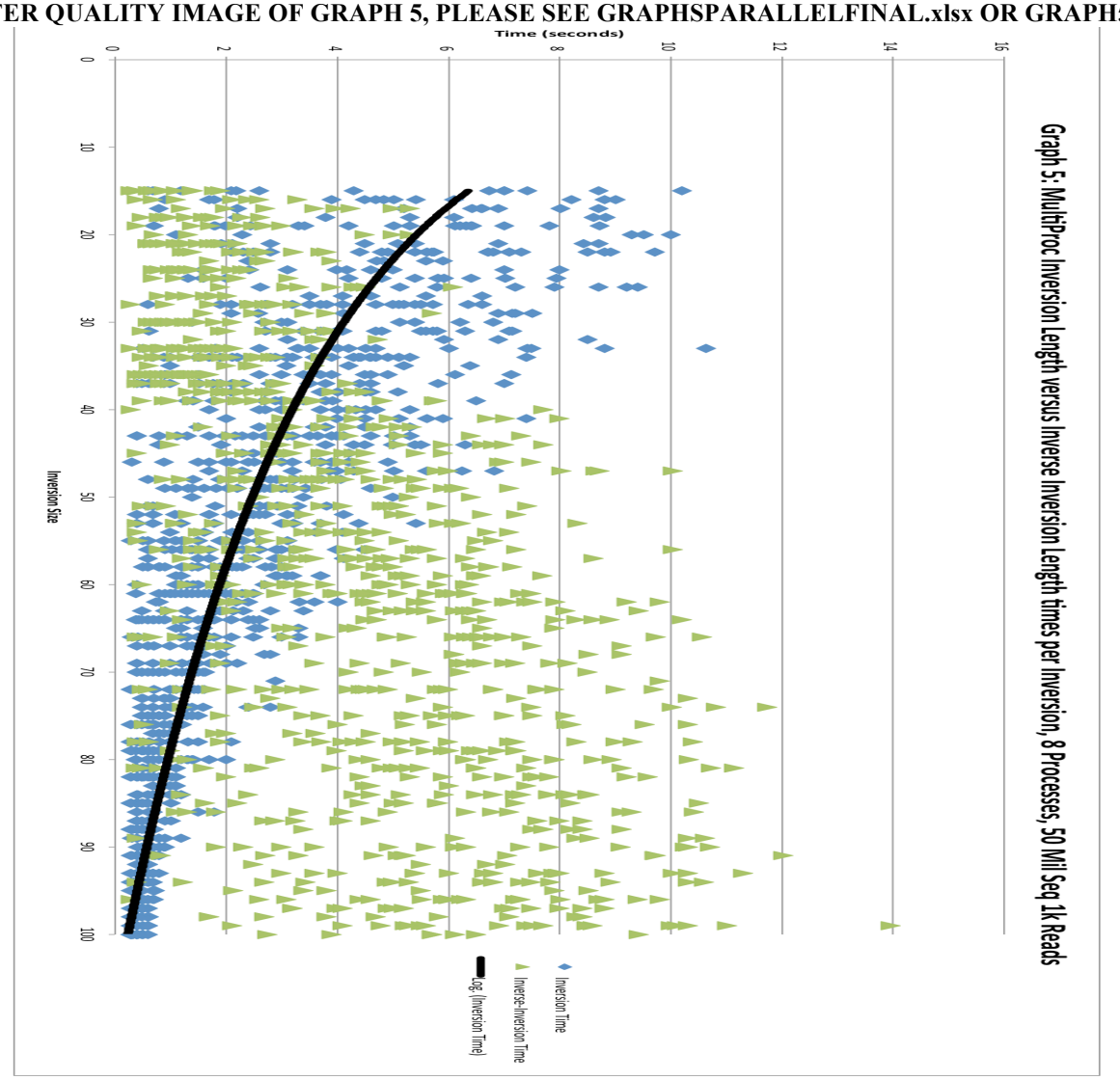
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017

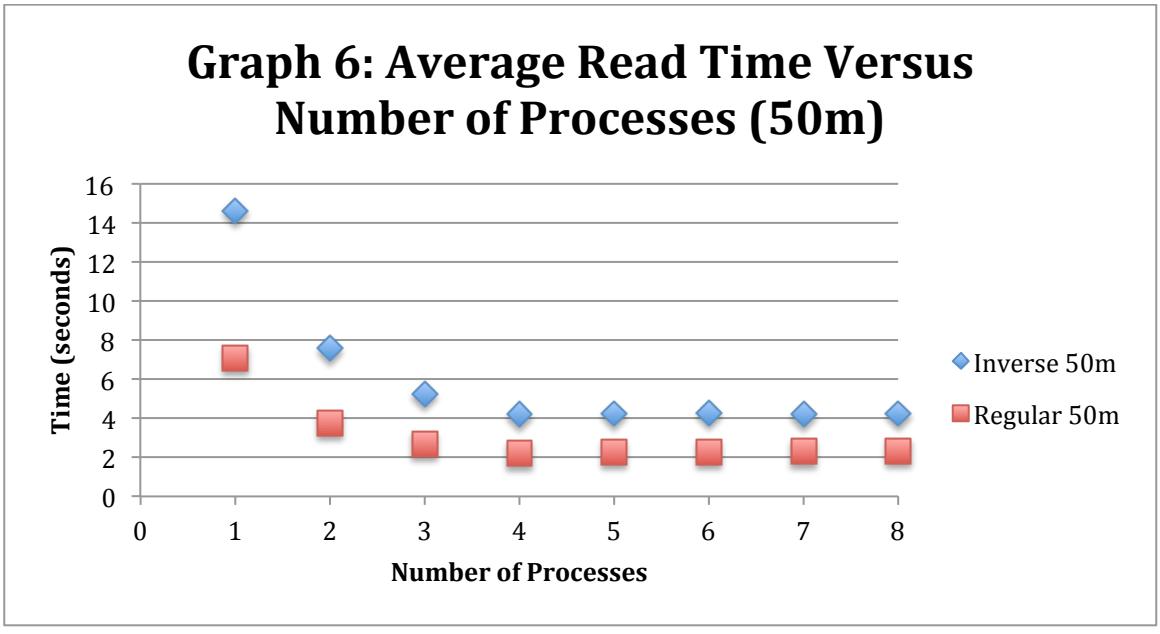




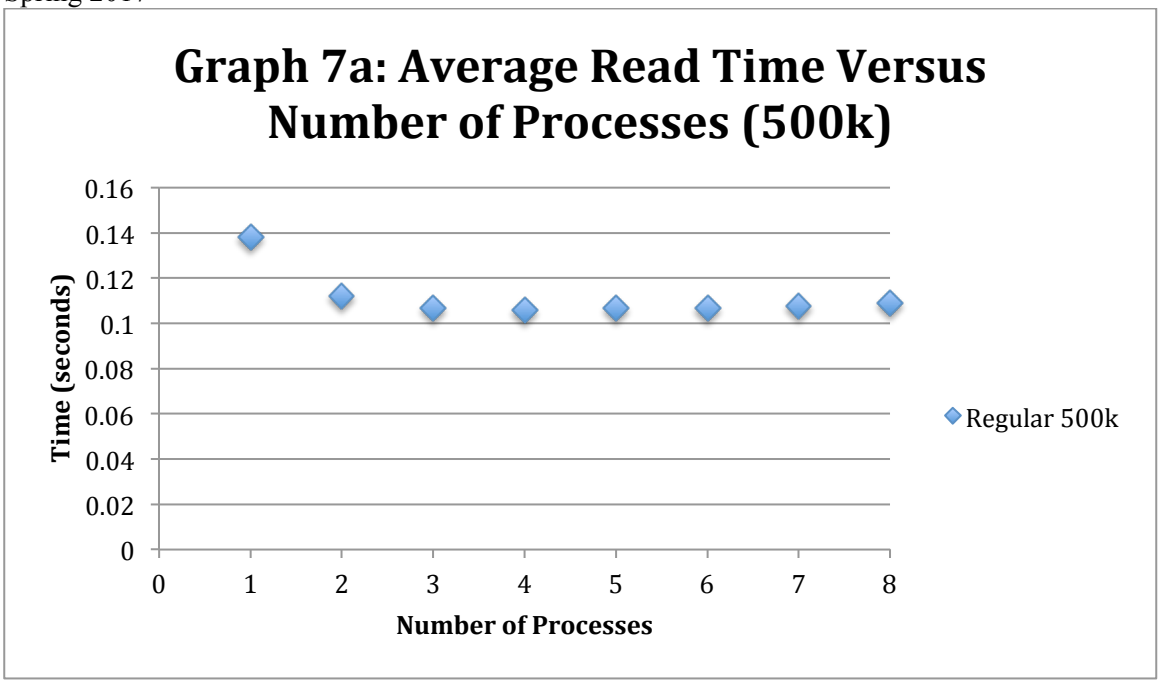
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017

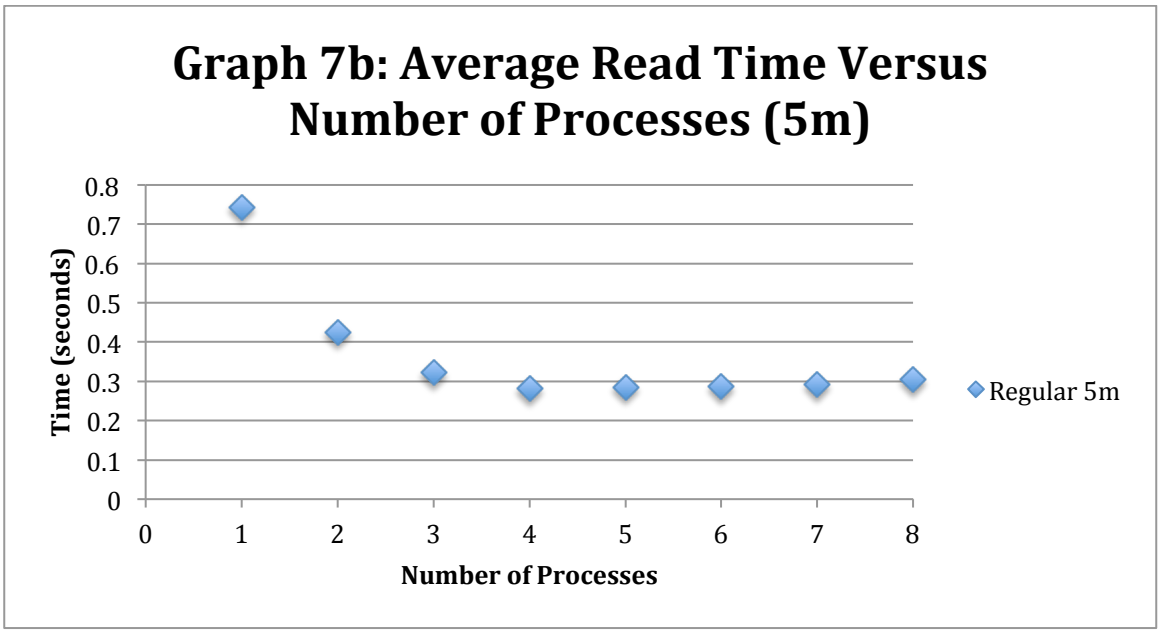
FOR A BETTER QUALITY IMAGE OF GRAPH 5, PLEASE SEE GRAPHSPARALLELFINAL.xlsx OR GRAPH5.png



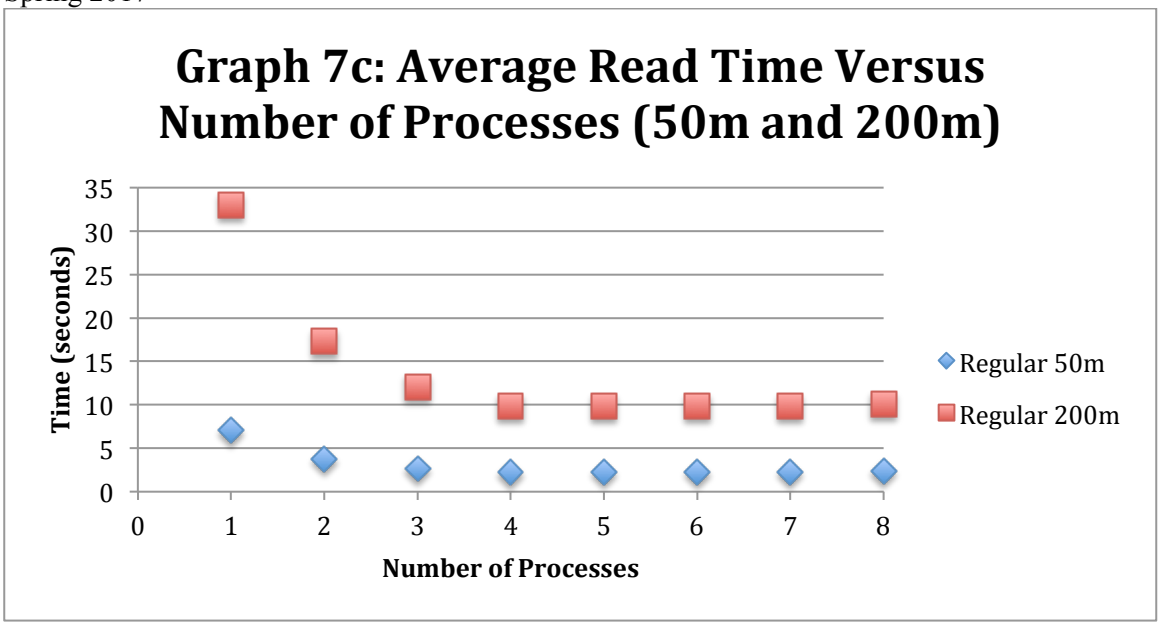


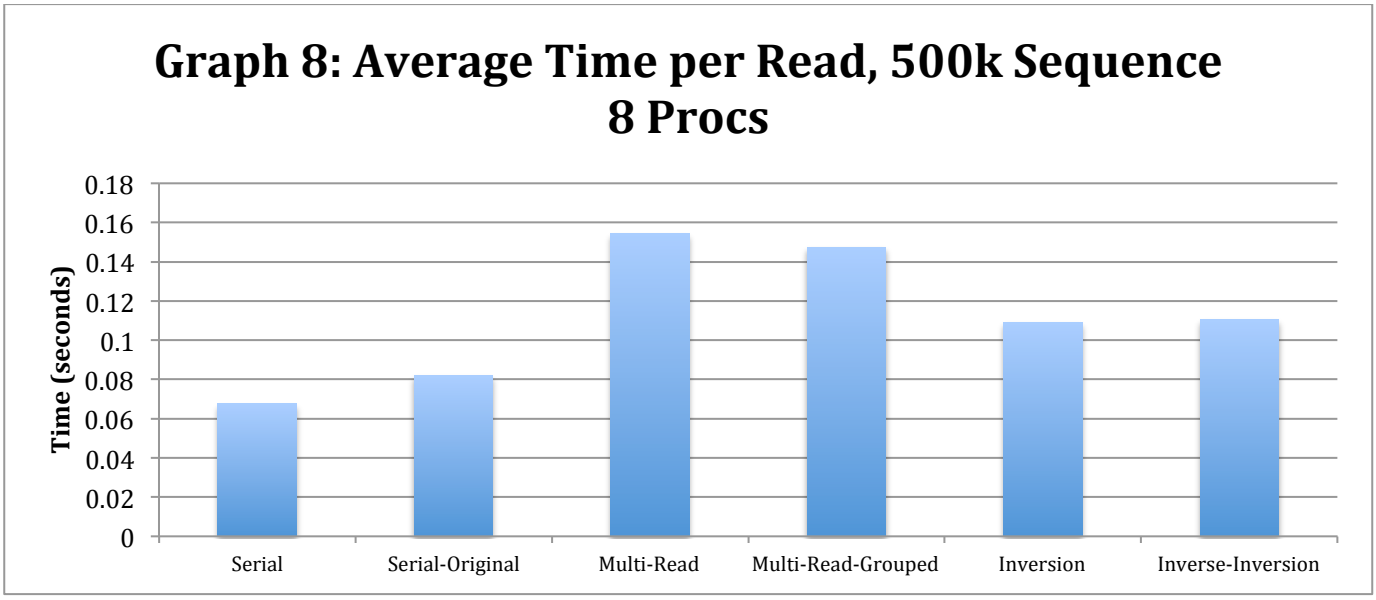
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017





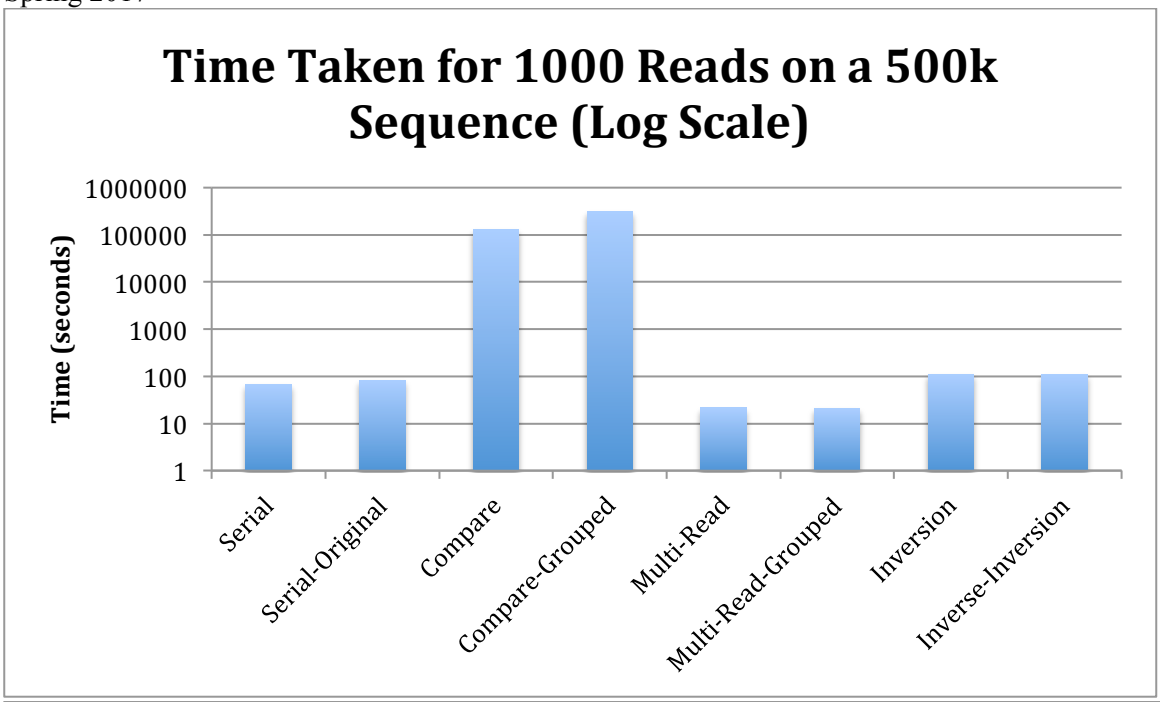
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017

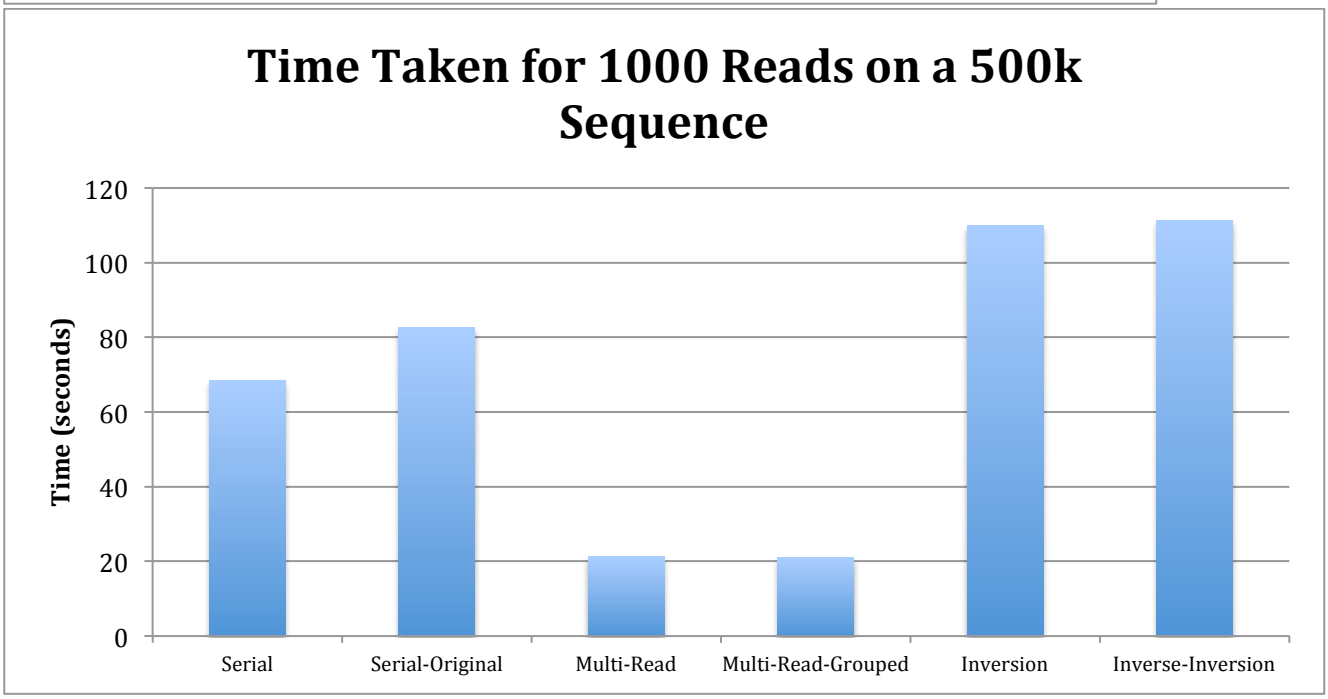




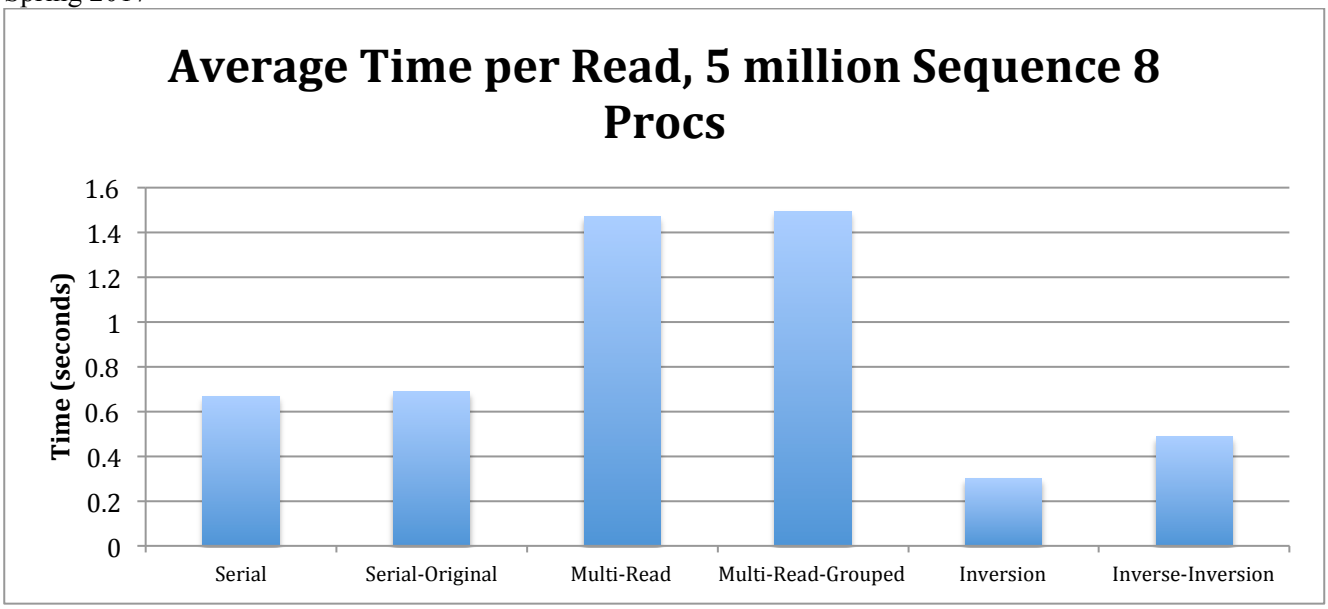
The rest of the graphs displayed were not explicitly mentioned in the report. Many are repeats of the same test but with total time taken instead of average time. Their titles are descriptive in their purpose.

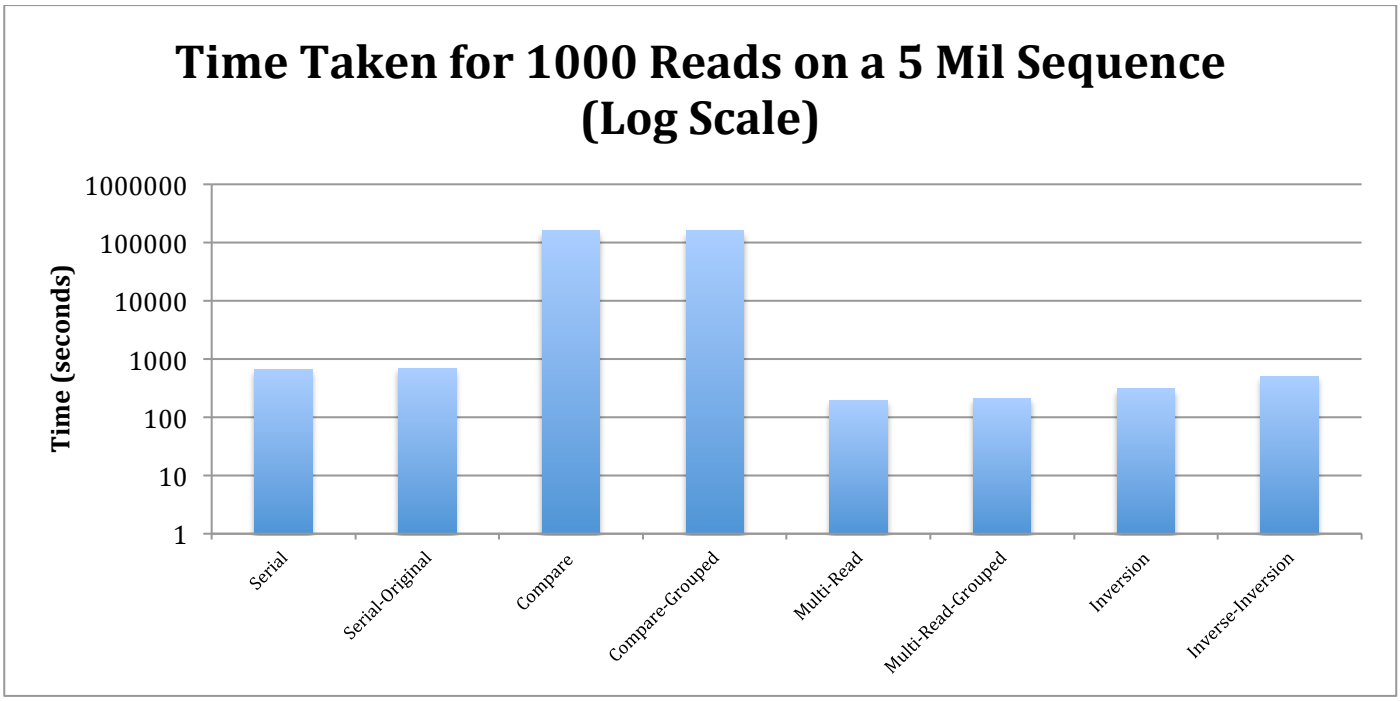
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017



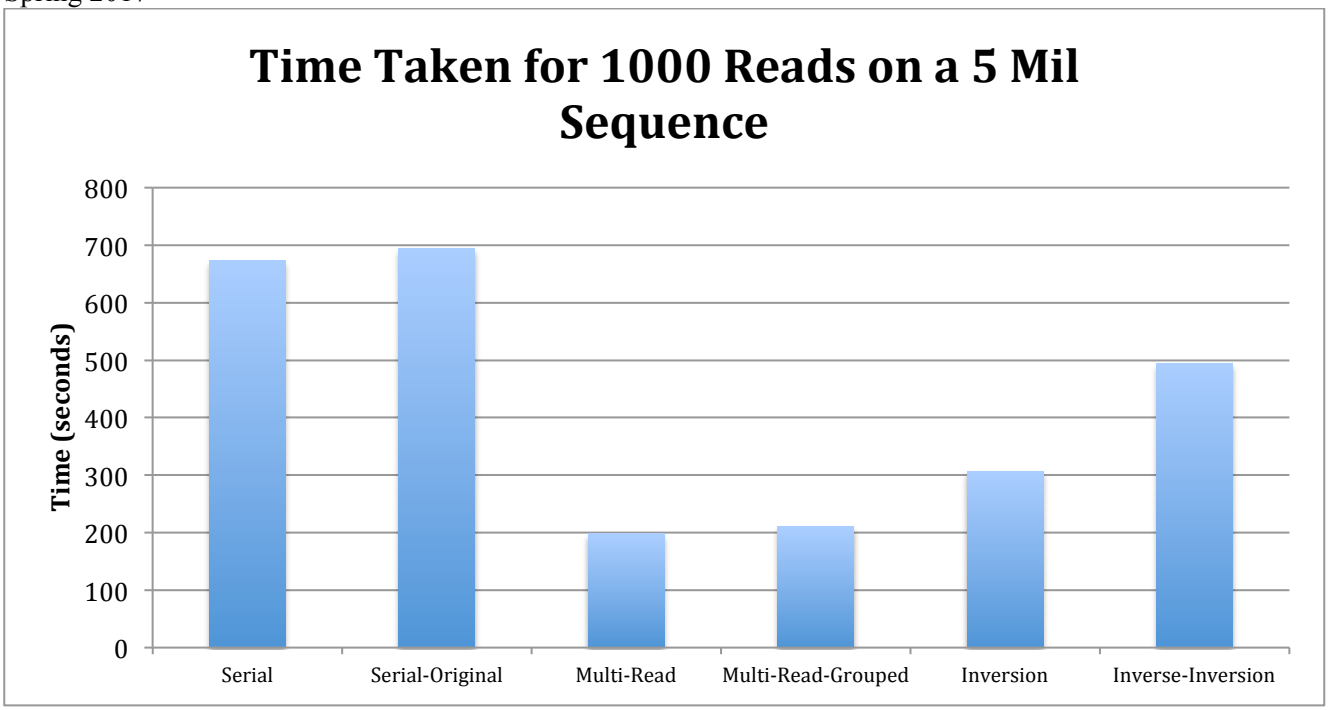


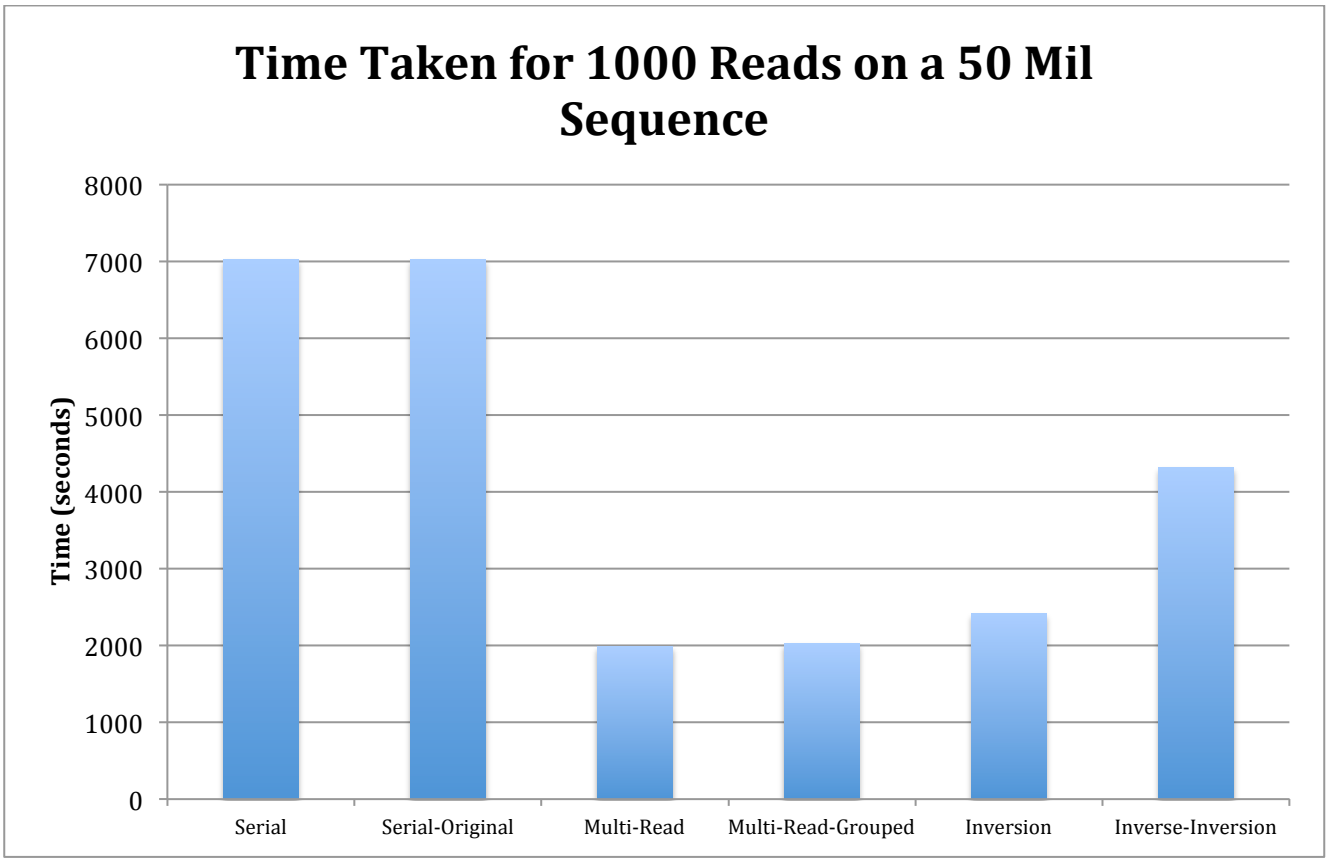
Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017





Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017





Joshan Bajaj – jbajaj1
Parallel Programming
Spring 2017