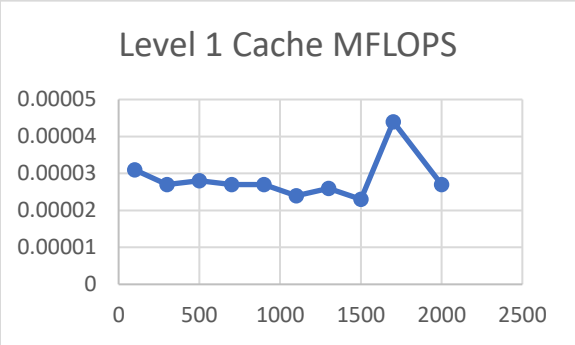
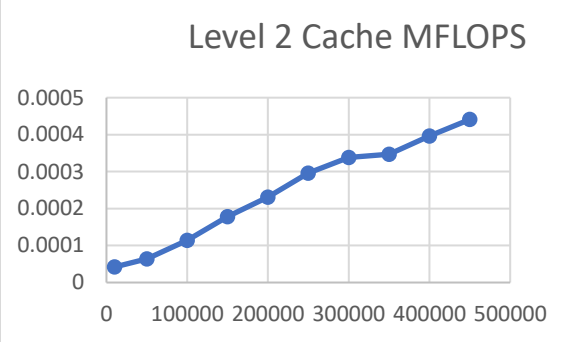
David Vondran

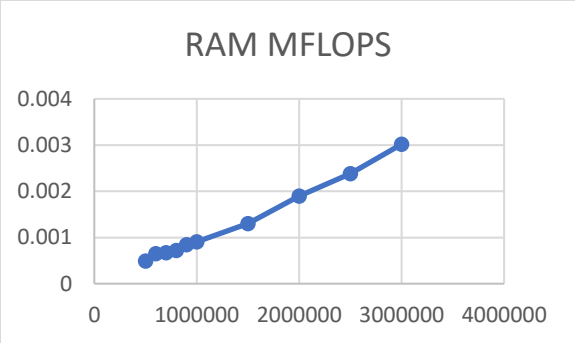
3/31/2022

Hw4

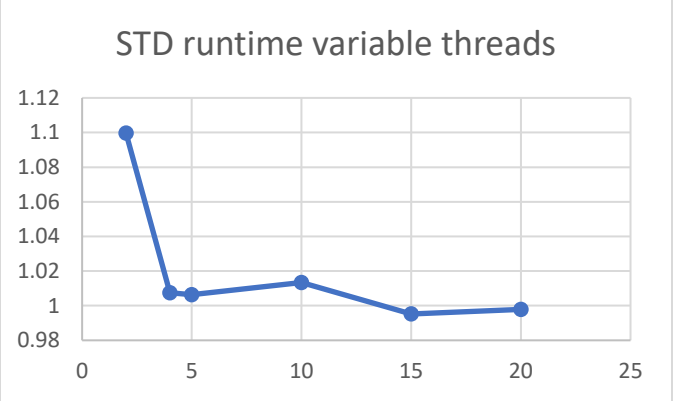
For this assignment, I was able to accomplish the tasks very well, especially considering that the task given was very similar in nature to that of homework 3, but with significantly simpler implementation of the code. I ran the implementation of the code on the standard ACI node of the cluster, which contains an Intel Xeon 2.8GHz processor. This processor utilizes L1 cache of 2x16 Kb and L2 cache of 2x2 Mb, meaning L1 cache values will hold 2000 8-bit values and L2 will hold 500000 8-bit values. This is the reason for my testing values. Utilizing pragma OMP, I was able to introduce threading in a significantly simpler manner than the multi-step process that was required for std threads to be implemented. For part 1, working on the std thread part of the assignment, I implemented 3 total parallel sections. One included finding the mean and the min and max, the second was utilized to find the std calculation, and the final was used in the threshold method to find the value c for building the threshold array. I could have found a way to implement threading on the final loop in this section, however I ran into difficulty on this part. For all of the parallel regions I implemented, I used P threads so that this number of threads could be changed depending on the desires of the user without having to change any hardcoded options. For the std method, I was able to get my threaded solution down to around .6 seconds, while the serial solution remained around 2.7-3 seconds. This is a significant decrease in the amount of time, and leads to significant savings over larger iterations. Given the 3 graphs shown below, we can see the algorithm runtime growth for the standard deviation. We can see that for the level 1 cache values, the runtimes are generally the fastest, hovering around .00003 seconds of execution time. This is because the level 1 cache ahs the fastest access values, meaning that the memory storage is able to be handled far quicker than it would for the larger values of N. For the level 2 cache, we are able to see that it grows linearly from the bottom values to the topmost. This is because the memory access time is staying generally the same, but the iterations are linearly increasing, increasing the overall runtime. The same can be seen for the RAM values, as they continually grow linearly for the foreseeable future values of N.



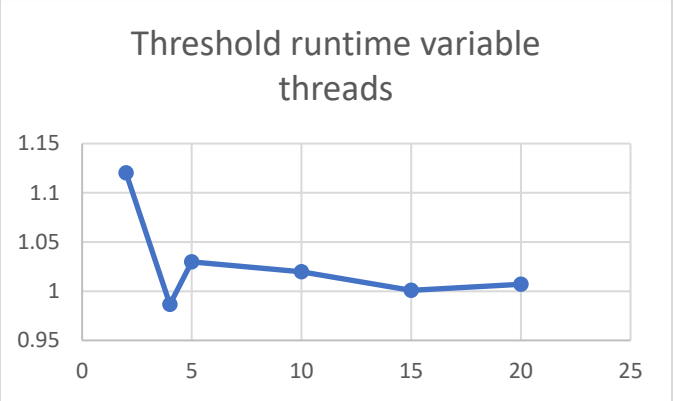




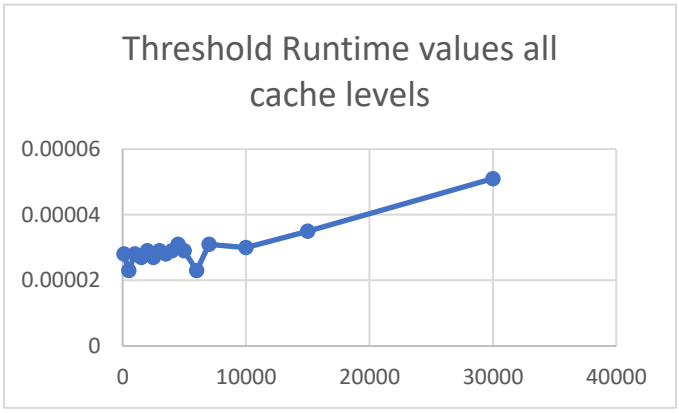
Beyond this, I was able to test the STD method with a variable amount of threads ranging from 2-20. These were chosen because number of cores (20) in the given processor. We are able to see that generally speaking, as thread count increased, runtime decreased. However, between 10 and 20 the benefits of more threads begin to rapidly fall off.



As for the threshold method, we were able to see a generally similar runtime chart with the best values coming from more threads, with a larger jump at the beginning.



For the threshold method, I was able to get the execution time down to around .9 seconds, while the serial version of this method took around 1.7 seconds. This is a significant decrease in execution time as well, cutting time almost in half. For the threshold method, we are able to see that the values of the runtime hold around .00003 until they reach 10000, where they begin to linearly increase. This is due to the increasing access time of the increasing levels of cache, as well as the amount of iterations that need to be completed over the values.



As for part 2, we can see a similar story unfolding to part 1. For this part of the assignment, I included one parallel region encompassing the entire code chunk, and a parallel for within the section encompassing the most outer for loop. I attempted to include further parallelization on the inner loops, however this was not allowed by the OMP package. However, even with only the outmost loop parallelized, I was able to get the standard run with N=2000 down from 18 seconds on the serial run to around 4 seconds on the parallel run. This was accomplished with P=10 processors. Below is shown a graph showing the run time of this algorithm with increasing values of N and the execution time. It grows exponentially as well with no distinguishable cutoffs between the cache values. The reason for this growth is more likely because the values of the iterations far outweigh the memory access values, and thus dictate the growth of the algorithm runtime.