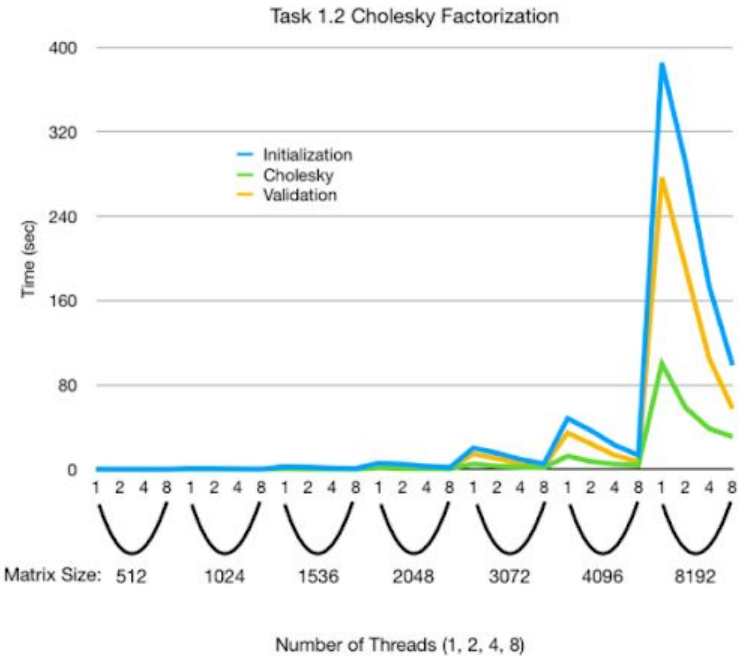
Nicholas Allaire, A10639753
Rammy Issa, A12471726
CSE 160: Parallel Computing
Programming Assignment #6
18 March 2018

Task 1 Writeup

openMP Directives:

For task 1 we decided to use the openMP for directive to parallelize the 3 parts of computing the cholesky factorization. We changed the scheduling from static, dynamic, and guided to find out which schedule works the best, and in our case we decided to use the dynamic scheduling with a block size of N * N * 0.1 for initialization and validation, and N * 0.1 for multT, where N is the matrix size. Also the initialization and the validation were able to make use of collapse, since each thread is given the same workload each iteration, whereas multT and cholesky where not able to make use of collapse.



Speedup for Cholesky ($S = T_{serial} / T_{parallel}$):

1024x1024:

1 thread (serial): 0.1729879s
2 threads: S = 0.1729879 / 0.09918809 = 1.744
4 threads: S = 0.1729879 / 0.08123994 = 2.129
8 threads: S = 0.1729879 / 0.06154895 = 2.811

2048x2048:

1 thread (serial): 1.460700s
2 threads: S = 1.460700 / 0.8334501 = 1.753
4 threads: S = 1.460700 / 0.6300302 = 2.318
8 threads: S = 1.460700 / 0.5073581 = 2.879

4096x4096:

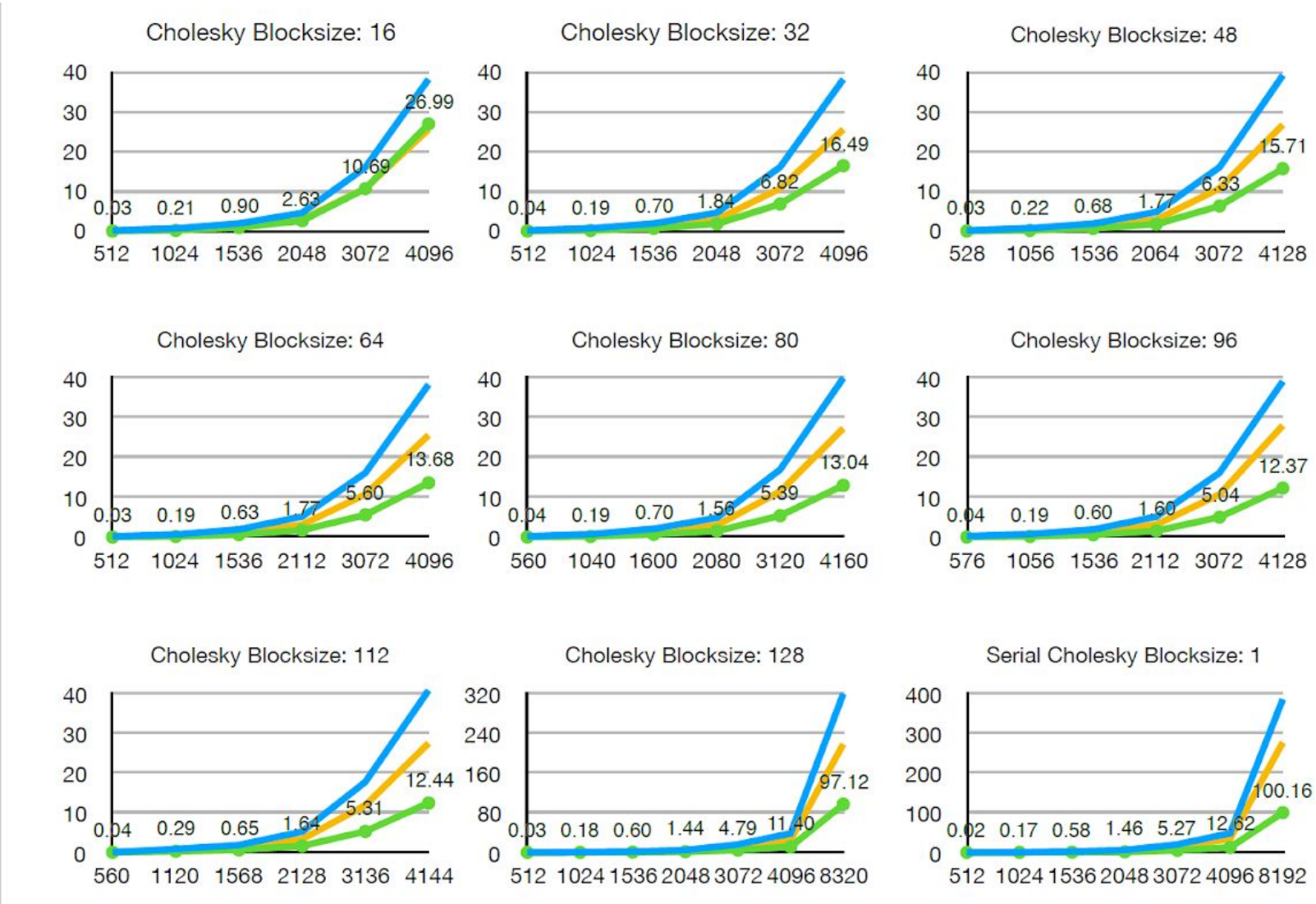
1 thread (serial): 12.62384s
2 threads: S = 12.62384 / 7.290255 = 1.732
4 threads: S = 12.62384 / 4.925299 = 2.563
8 threads: S = 12.62384 / 3.822506 = 3.303

8192x8192:

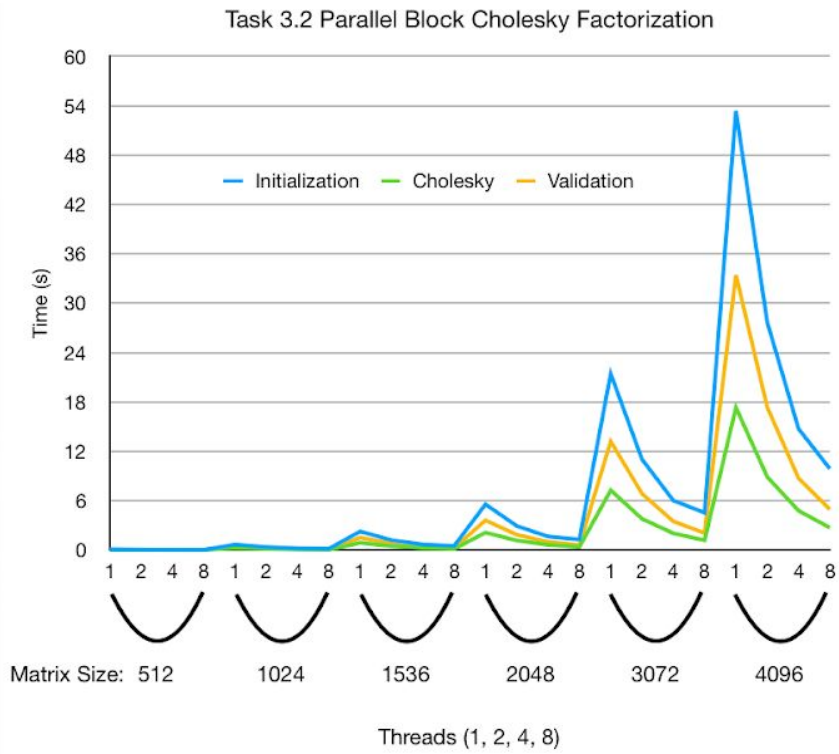
1 thread (serial): 100.1645s
2 threads: S = 100.1645 / 58.56536 = 1.710
4 threads: S = 100.1645 / 38.70444 = 2.588
8 threads: S = 100.1645 / 30.80954 = 3.251

Task 2 Writeup

After running through many different tests, our results showed that a block size of 128 was ideal for computing the block cholesky factorization. We were able to generate a speed up of $\frac{12.624}{11.401} = 1.107$ for a matrix size of 4096 x 4096. Also a block size of 16 had the worst results of the block sizes tested, and as the block size increased the cholesky factorization became faster and faster. This can be attributed to the size of the cache line, meaning as the block size got larger it filled the cache line enough to minimize the amount of cache misses. The program would then need to go into memory and retrieve data, which is an expensive operation.The size of the matrix only started to affect the results once it started to get large, at around 3072 x 3072, and the time from 2048 to 3072 almost tripled. Below are the results of each block size, where each graph represents a specific block size and the lines represent the timings for initialization, cholesky factorization, and validation for matrices sizing from roughly 512 to 4096. The blue = initialization, green = cholesky factorization, orange = validation and the x-axis represents the matrix size (N x N) and the y-axis represents time (s). The data points represent cholesky factorization timings for each matrix size for easy comparison.



Task 3 Writeup



Speedup And Efficiency When Serial and Parallel Block Sizes Match For Best Speed

Matrix Size	Blocksize	Threads	Serial Time	Parallel Time	Speedup	Efficiency
2048	128	2	2.131786	1.164619	1.83	0.915
3072	128	2	3.800243	2.124533	1.79	0.895
4096	128	2	17.32218	8.887285	1.95	0.975
4096	128	4	17.32218	4.785665	3.62	0.905

openMP Parallelization Strategies:

For parallelizing blockCheolesky, we tried a number of approaches. When beginning to parallelize, we did not run into any bumps. We were able to parallelize the initialization with no problem. This was because we implemented perfectly nested loops in our initialization.However it was not so obvious how to implement parallelization for our blockCholeksy factorization. This was because in factorization we did not have perfectly nested loops. OpenMP does not support parallelization for not perfectly nested loops. To get around this and parallelize our factorization, we had to parallelize each loop on its own. To calculate L11, there wasn't any problem parallelizing that case. For L21 and L22, we tried different approaches. First we tried to parallelize with shared variables. This did not work because the iteration variables will have

different values, causing threads to execute on the same blocks. After this, we had to learn how the iteration indexes were being used by each thread. After learning this we were able to make the correct variables private to each thread. This gave the threads the correct variables that other threads could not change. We also needed to test and how much we should give each thread. After testing and determining amount of work each block should get, we were able to get great speedup. After completing L21, We used the same approach for L21.