In this week's project, I learned how to use OpenMP to get a parallel version of prefix sum and improved the parallel matrix multiplication.

First of all, I learned how to use OpenMP and implemented prefix sum with it. The main idea of parallel prefix sum was to divide the vector into several chunks with equal size. The number of chunks could be determined by the number of threads. Then, let each thread add up the prefix sum in each chunk and store the sum of the last element in the chunk to a vector so that other chunks could add the sum prior to their chunk. This required every thread to stop until all the threads were done so that the prior sum would not be missed. This could be done by “barrier”. After that, divide the vector into the same chunks as before, which needed some help from schedule(static) to make sure the chunks before and after stopping were the same. Finally, let each chunk add its prior sum to get the prefix sum of the whole vector.

After implementing the parallel prefix sum algorithm, I tested it with different sizes of vectors. Here is the time diagram. The y-axis is the running time of sequential and parallel algorithms. The square of the (60 \* x-axis) is the size of the vector, which means that the maximum size of the vector I used is 12110400.

As you can see from the graph, when the size of the vector is small, the sequential prefix sum algorithm is faster than the parallel prefix sum algorithm. As the size of the vector increases, the speed of the parallel prefix sum algorithm becomes much faster than the sequential prefix sum algorithm.

In addition to the prefix sum algorithm, I also worked on parallel matrix multiplication algorithms. I tried to modify the traditional matrix multiplication code to make it cache-friendly to increase its time efficiency. First of all, I tried to swap j and k so that there was more multiplication within a row, and the cache could help to reduce the running time. However, when the size of the matrix is very large, it is not efficient even to do multiplication in the same row. Also, for each loop, I still need to move down to the column. This really wastes lots of cache. Therefore, I did some research online and learned a technique called matrix block, which can help with the cache problem. To be more specific, the matrix block breaks a big matrix into small blocks and does matrix multiplication within the blocks. Then, add up each block to get the final result of the multiplication. This is much more cache-friendly than the previous version since doing matrix multiplication within a small matrix can be more cache-friendly. However, the size of the block should satisfy the requirements of matrix multiplication. Therefore, I only tested on the matrix with the perfect square size so that I could divide the block with the same size. In addition, I also replaced for loop with the memset to increase the time efficiency of the algorithm.

After that, I tested the matrix with the traditional parallel matrix multiplication, the cache-friendly parallel matrix multiplication, and the matrix block parallel matrix multiplication algorithms. Here is the running time diagram. The y-axis is the running time of the algorithm. The square of the x-axis is the size of the matrix.

As you can see from the graph, as the size of the matrix increases, the traditional parallel matrix multiplication algorithm becomes slower. The second slowest algorithm is parallel matrix multiplication with the swap of j and k. But it is really close to the other two algorithms. Finally, the parallel matrix block with and without the memset is the fastest. They are very close and almost the same.

In conclusion, parallel algorithms and cache can play an important role in improving the time efficiency of the algorithm. My computer is not powerful enough to handle the huge size of vectors and matrix, but the general trend showed that parallelism and cache is important. Accordingly, as for large samples, using parallelism and cache-friendly algorithms can really help to improve the time efficiency of the program.