Program 1: Dense Matrix Multiplication

I went about program 1, by creating two new matrix-matrix multiplication functions in matmult_omp.c. The matrix is mat_ mult_no_tiling(). This function is essentially a parallelized version of mat_mult(). The part that needs to be parallelized is the part containing multiple iterations and mathematical calculations. In this case, there are nested for loops indexing the matrices. Serially executed, like in mat_mult(), would mean that each iteration through the nested for loops would occur m*p*n times one by one. It would be faster to run all the iterations of the nested for loops concurrently. So, I used the parallel for directive in OpenMP to parallelize these nested for loops. I specified a "#pragma omp single" directive region so that the execution end time ("end = omp_get_wtime();") would not be duplicated by multiple threads but rather a single thread. Timing operations can also have some overhead which can add up if used by a lot of threads, thus the single directive reduces such overhead.

The second function I wrote, mat_mult_tiling(), implements the technique of tiling to multiplying matrices faster. It takes smaller sized matrices from each input matrix and computes the product of these smaller matrices. Then, to get the final matrix the product of the smaller matrices are added up accordingly. As seen in the code, my implementation of tiling involves six nested for loops and three of which have loop dependency because the starting value is of one of the for loops before it. Similar to mat_mult_no_tiling() I utilized the parallel for directive again. This time I include the collapse(int) clause to combine multiple loop levels into a single loop. By collapsing the loops, you can parallelize the combined loop, which can potentially lead to better load balancing and improved performance.

In both functions I included shared() and private() clauses to specify how variables should be treated within parallel regions. shared() specifies that the variable function should be shared among all threads in the parallel region. So, matrix C should be shared because C is a cumulative dot product of all threads and iterations. In contrast, the private() clause ensures that the variables should be private to each thread and have its own separate copy. It is used to prevent data races when multiple threads need to use temporary variables.

I used an .sh file to automate the execution of the program with varying command line arguments. The program outputs a "out.txt" file containing the execution time for the serial function, parallelized function, and the parallelized tiling function. used to multiply matrices of three different dimensions. Parallelization was tested using 1, 2, 4, 8, 12, 14, 16, 20, 24, and 28 threads. The results appear in the output text file as follows:

```
NUM OF THREADS: #
MATRIX SIZE: A[nrows][ncols], B[ncols][ncols2], C[nrows][ncols2]
Serial execution time (#  thread): t seconds
No tiling parallel execution time (# threads): t seconds
Tiling parallel execution time (# tiles): t seconds
```

Taking the execution time from the out.txt file, I used excel to create graphs showing speed up of the three different matrix sizes.

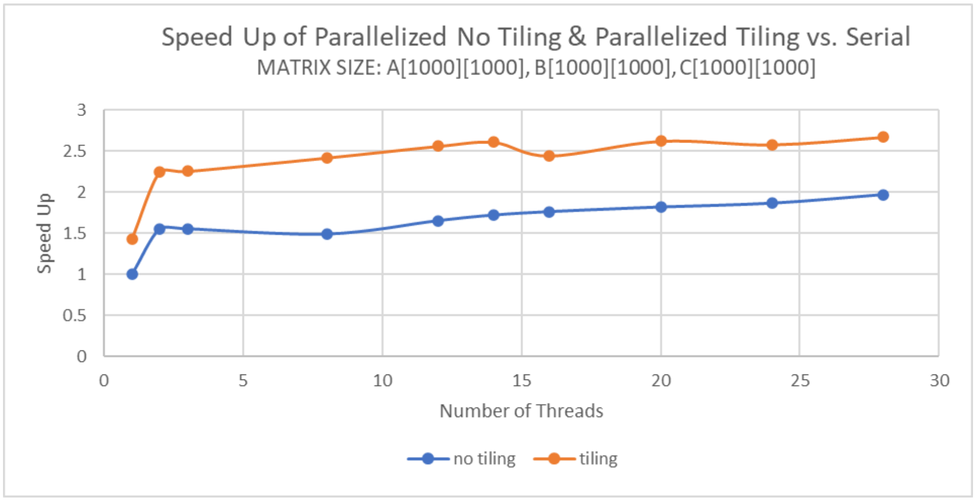

Figure 1.

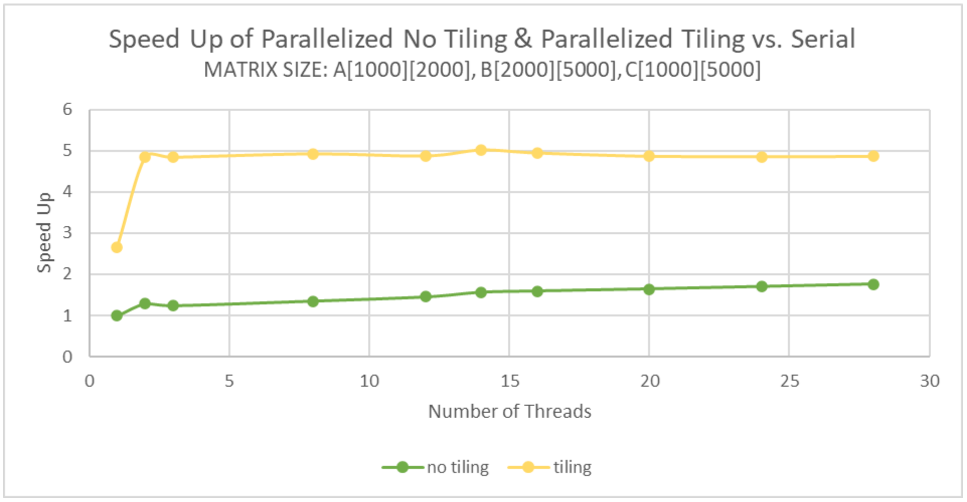

Figure 2.

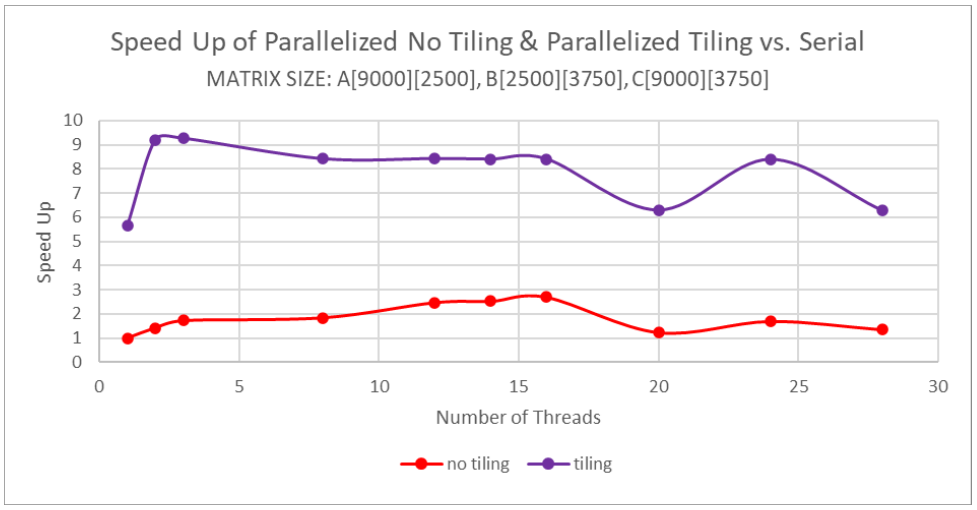

Figure 3.

Generally, the greater number of threads decreases the runtime because the more threads there are, the less amount of data is processed per thread and thus the process per thread is quicker. However, my results in Figure 3 show that at 20 threads for the matrix size A[9000][2500], B[2500][3750], and C[9000][3750] the speed up decreased compared to 16 threads, which means runtime increased. This may have occurred because of data dependencies where the for-loops iterations have dependencies on the previous loop and there are larger amounts of threads waiting for the dependencies to resolve.

The larger the size of the problem, the longer the runtime. The size of the problem influences the granularity of work. There is more work to distribute among the threads. According to my results, it is likely that increasing the number of threads will also increase the performance.

The performance characteristics are not consistent for the different shapes of the matrices. For example, in Figure 3 matrix A has a substantially larger number of rows compared to columns while matrix B's dimensions are more similar to each other. The speed up of matrix matrix multiplication of this size is not constant. The efficiency of parallelization was affected by the fact that matrix A is long and thin while matrix B is more square. This may introduce load balancing issues.