Parallelization of the Matrix Multiply Algorithm

1st Sean Tronsen
Department of Computing
Coastal Carolina University
Conway, South Carolina

I. Introduction

The introduction of parallelism in the field of computing increased the speed at which programs could execute on a given set of hardware, given that their instructions took advantage of parallel processing. Matrix multiplication is a relatively simplistic computational task that is also well-known for its slow execution speed. This occurs because the standard algorithm for doing so (discussed later in the text) has a big O of n^3 for its time complexity. In order to get around this common slowdown, we introduced parallelism into the solution and observed the speedup that occurred as a result. All testing and data collection were performed on XSEDE's Expanse computing cluster.

II. Background

Matrix multiplication is a concept that comes from linear algebra and the core computation relies on another concept which is referred to as the dot product. The dot product refers to the process of multiplying a row from the first matrix, matrix A, by the column of the second matrix, matrix B. The action produces a singular value, being the element stored at the index related to the row and column involved in the calculation of the dot product. A code segment illustrating this series of calculations is shown below in the C programming language.

```
void serial_matrix_multiplication(double *A,
double *B, double *C, int M, int N, int K)
{
    double my_sum;
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            my_sum = 0;
            for (int k = 0; k < K; k++) {
                my_sum += A(i, k, K)
                * B(k, j, N);
            }
            C(i, j, N) = my_sum;
        }
    }
}
```

To detail the function shown, three pointers to double-precision floating-point matrices are passed in alongside three integer variables used to denote the length of each's rows and columns. It should also be noted that the referenced and manipulated in a single-dimensional format, or in other words, the function assumes that the programmer has passed in three matrices flattened into arrays.

This code segment was parallelized to decrease overall execution time using the OpenMP library which is a parallelization library written for the C, C++, and Fortran programming languages. Its use drastically simplifies the process and reduces the complexity typically involved in parallelizing code segments using operating system process/thread APIs. Whereas the programmer normally must keep track of each and every thread, using OpenMP allowed us to achieve the same effects oftentimes with the addition of only one compiler directive. In the C++ programming language, such directives are referred to as pragmas. An example of one such pragma is shown below.

```
#pragma omp parallel for
```

III. Parallelization

Several options existed when it came to parallelizing the original serial matrix multiplication algorithm, but the most notable was the parallelization of either the outer loop (rows) or the inner loop (columns). Both of these were briefly tested, but it was quickly learned that column parallelization made better use of the target hardware available on the Expanse cluster. The code segment below illustrates how the original serial segment was parallelized.

```
void parallel_matrix_multiplication(double *A,
double *B, double *C, int M, int N, int K) {
    for (int i = 0; i < M; i++) {
#pragma omp parallel for
        for (int j = 0; j < N; j++) {
            double my_sum = 0;
            for (int k = 0; k < K; k++) {
                my_sum += A(i, k, K)
                * B(k, j, N);
            }
            C(i, j, N) = my_sum;
        }

    }
}
```

One may quickly observe that the only true change was the introduction of the OpenMP pragma which was all that was required to parallelize the original segment. Now that parallelization has been achieved, three additional concepts

come into play and these are parallel runtime, speedup, and overall efficiency.

$$ParallelRuntime = P_r = T - T_s$$

$$Speedup = S_p = \frac{T_1}{T_p}$$

$$Efficiency = E_p = \frac{S_p}{p}$$

In the above equations, the variables T and p refer to time and threads/processors respectively. Using a series of runner programs which will be detailed in the Experimentation section, sets of data were generated from each of these equations to model the performance of the algorithms tested.

IV. Algorithms

Before continuing on about our testing, we must first note that more than one algorithm was at play during this study. The standard matrix multiplication algorithm was detailed above in both parallel and serial formats. In addition to this, the DGEMM algorithm was also tested to determine the effect on which hardware caching would have on processing speed on the target hardware. The algorithm aims to improve upon the temporal locality of the computation as the spatial locality is already exhibited through row-major traversal. The improvement in temporal locality occurs because blocking operations rapidly access the contents of blocks within a matrix, thus increasing the chances that said block remains in the cache throughout the operations required on it.

V. Experimentation

All algorithms were implemented in the C++ programming language and tested on the XSEDE via the EXPANSE portal. Execution time and gigaflop data were collected within a series of CSV files generated by associated runner programs (also referred to as sweep programs). Each of these programs ran as a singular job on the cluster. The accuracy of each implementation's computation was verified against the original using a custom command-line utility, titled my_diff. This utility program examined the contents of the resulting matrices output by all versions of the algorithm and compared each for equality. In addition, the utility also calculated the total sum of the squared error, TTSE, and the average percent relative error, AVGPRE.

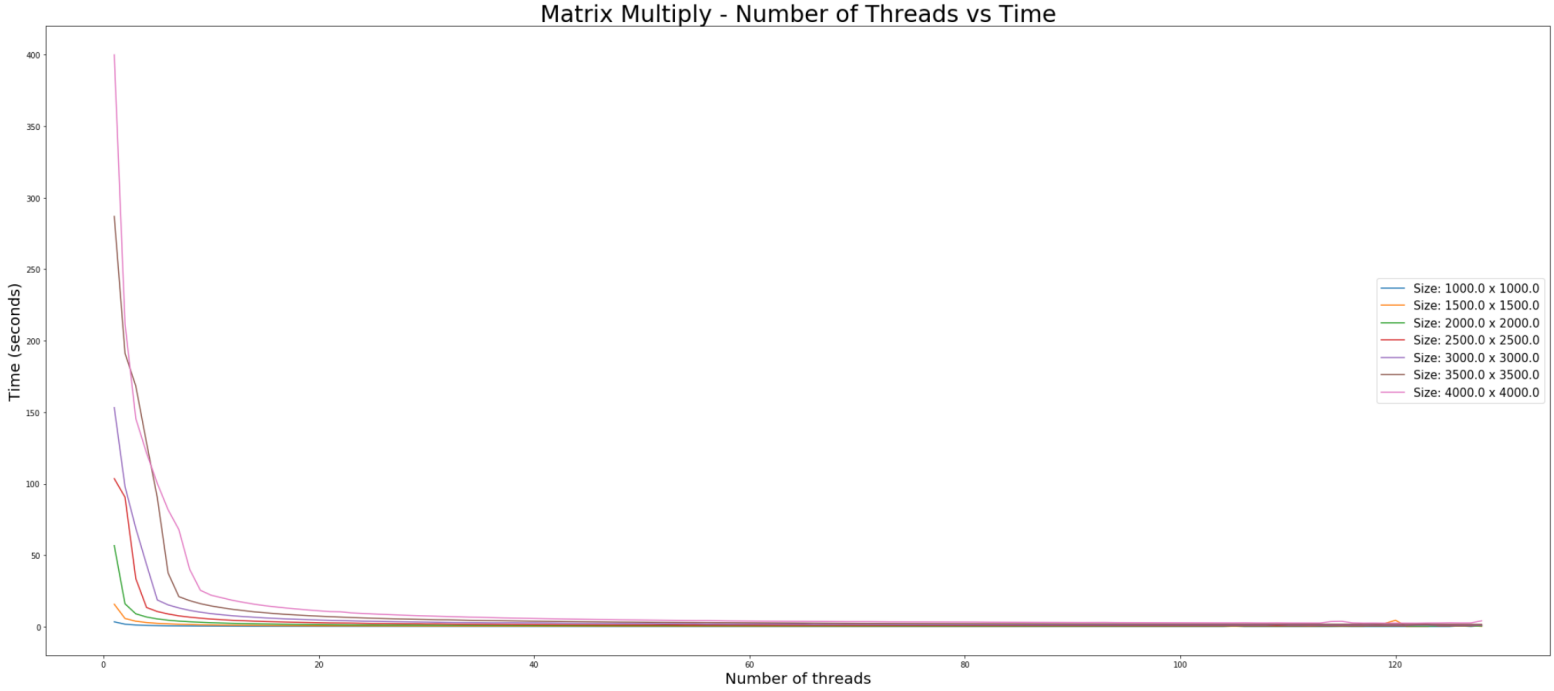
$$TSSE(A, B) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (A[i][j] - B[i][j])^2$$

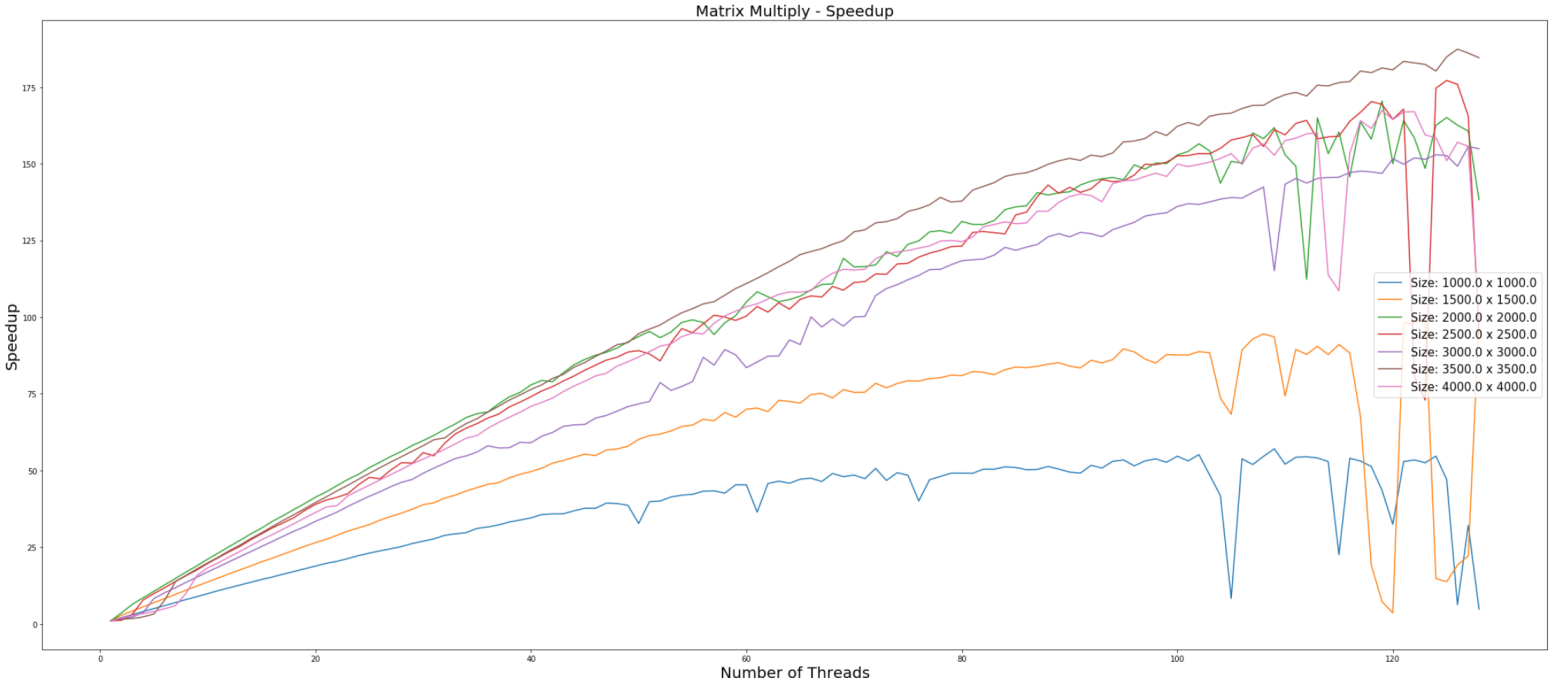
$$AVGPRE(A, B) = \frac{1}{M * N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} |\frac{A[i][j] - B[i][j]}{A[i][j]}|$$

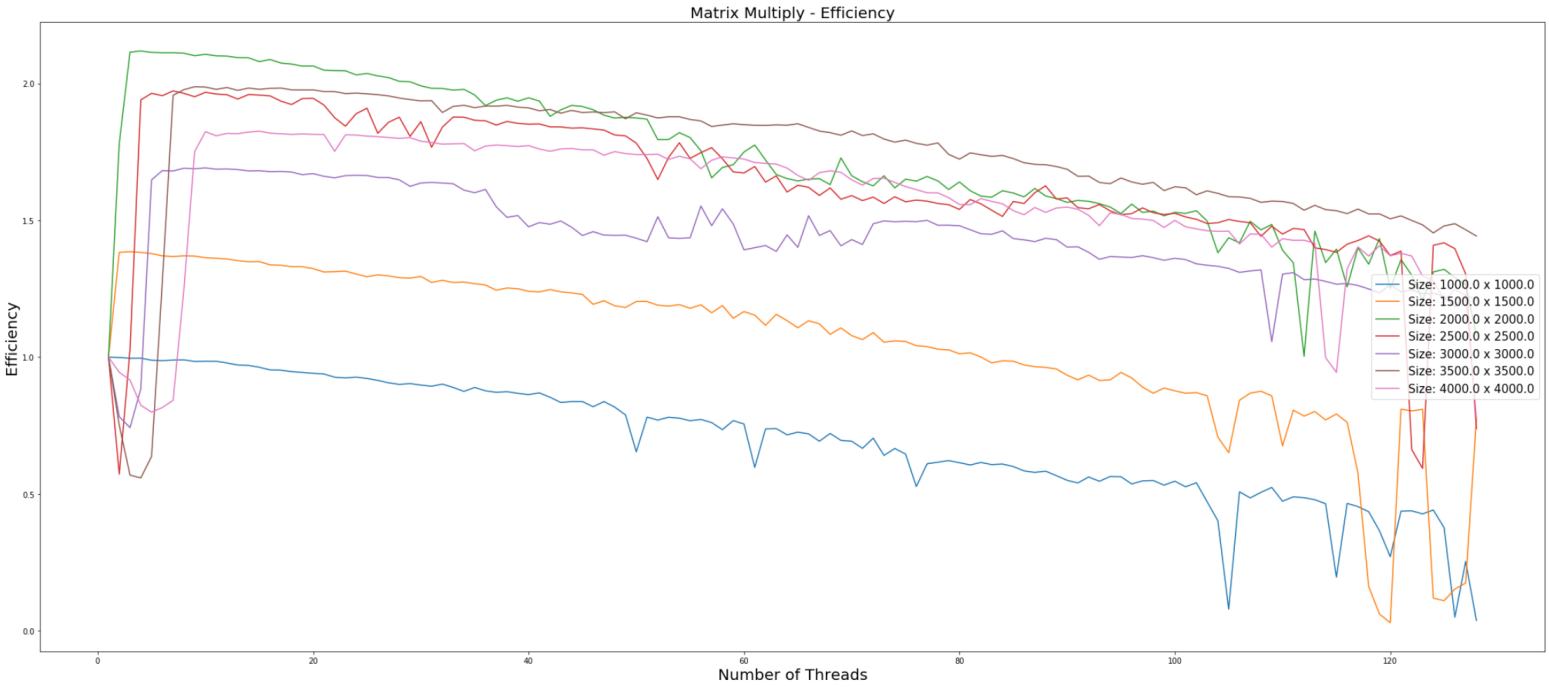
Lastly, the collected data was analyzed and displayed using MatPlotLib within a Jupyter Notebook running a Python 3 kernel.

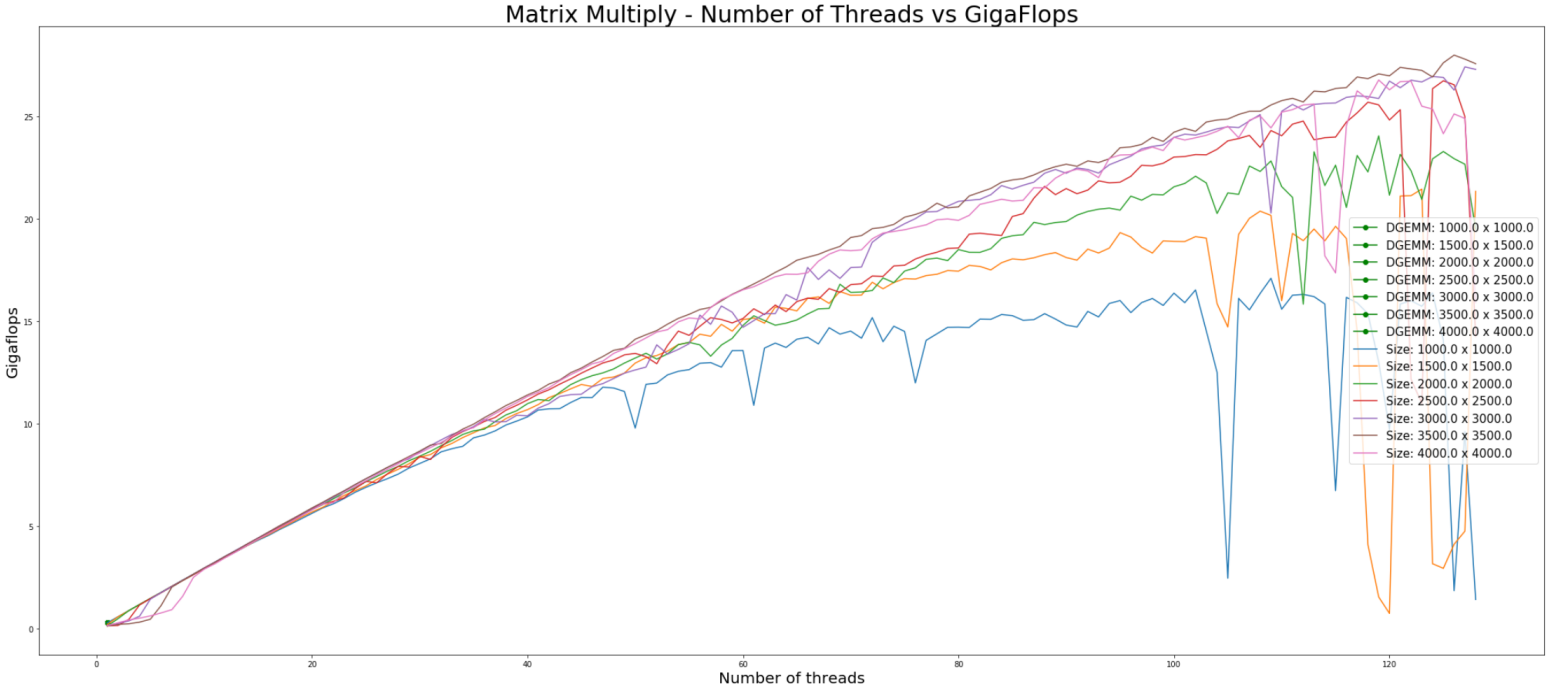
VI. Results

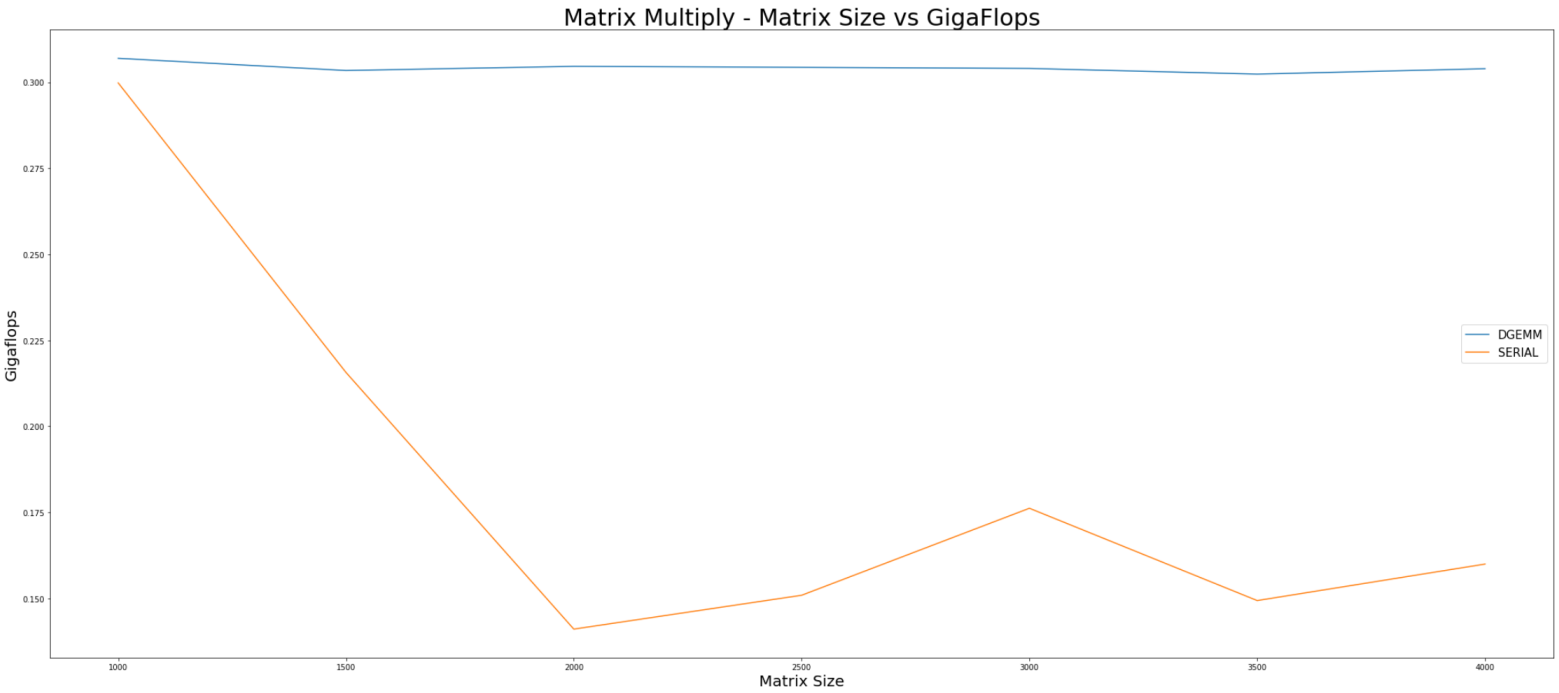
The results from the experimentation illustrated the performance benefits of parallelizing the original serial matrix multiply algorithm through the aforementioned metrics of runtime, speedup, and efficiency. In viewing the graphs on the following page, it can be seen that as the number of threads increases, the overall runtime decreases regardless of matrix size. However, in viewing this it is also quickly apparent that a limit exists to this benefit which implies that throwing an infinite amount of threads at the computation is a poor use of the hardware. Notice that as the graph proceeds to the right, the lines all begin to merge despite the different matrix sizes. This is corroborated by both the graph for speedup as well as the one for efficiency. As the limit of the runtime is approached, it can also be seen that the efficiency of each thread simultaneously is decreasing. Although a speedup is still occurring, the pattern denotes that cost will eventually outweigh the benefit of parallelism after a certain number of threads has been reached.


Matrix Multiply - Number of Threads vs Time
Time (seconds)
Number of threads
Size: 1000.0 x 1000.0
Size: 1500.0 x 1500.0
Size: 2000.0 x 2000.0
Size: 2500.0 x 2500.0
Size: 3000.0 x 3000.0
Size: 3500.0 x 3500.0
Size: 4000.0 x 4000.0


Matrix Multiply - Speedup
Speedup
Number of Threads
Size: 1000.0 x 1000.0
Size: 1500.0 x 1500.0
Size: 2000.0 x 2000.0
Size: 2500.0 x 2500.0
Size: 3000.0 x 3000.0
Size: 3500.0 x 3500.0
Size: 4000.0 x 4000.0


Matrix Multiply - Efficiency
Efficiency
Number of Threads
Size: 1000.0 x 1000.0
Size: 1500.0 x 1500.0
Size: 2000.0 x 2000.0
Size: 2500.0 x 2500.0
Size: 3000.0 x 3000.0
Size: 3500.0 x 3500.0
Size: 4000.0 x 4000.0


Matrix Multiply - Number of Threads vs GigaFlops
Gigaflops
Number of threads
DGEMM: 1000.0 x 1000.0
DGEMM: 1500.0 x 1500.0
DGEMM: 2000.0 x 2000.0
DGEMM: 2500.0 x 2500.0
DGEMM: 3000.0 x 3000.0
DGEMM: 3500.0 x 3500.0
DGEMM: 4000.0 x 4000.0
Size: 1000.0 x 1000.0
Size: 1500.0 x 1500.0
Size: 2000.0 x 2000.0
Size: 2500.0 x 2500.0
Size: 3000.0 x 3000.0
Size: 3500.0 x 3500.0
Size: 4000.0 x 4000.0


Matrix Multiply - Matrix Size vs GigaFlops
Gigaflops
0.300
0.275
0.250
0.225
0.200
0.175
0.150
1000
1500
2000
2500
3000
3500
4000
Matrix Size
DGEMM
SERIAL

In addition to the previously mentioned metrics, the gigaflops performed by the algorithms were also recorded and analyzed. As expected, the value of this metric increased when the number of threads allotted to the program was increased. An interesting result of the testing though was the fact that for the serial execution of the DGEMM algorithm, the gigaflops recorded remained relatively constant while they plummeted for the standard algorithm.

VII. Conclusion

Implementing parallel programs on multi-core systems has become a necessity in the modern-day. This is because the focus has shifted away from making the serial case fast via the hardware to how the serial case could have been improved through parallel programming. The parallelization of the matrix multiply algorithm confirms this as it demonstrates that increasing the number of threads lowers the program runtime, to an extent. An efficiency wall will eventually be hit depending on the hardware and the algorithm, but up until that point increasing the level of parallelism greatly reduces overall runtime. While the majority of this text focuses on the standard algorithm, it is unclear to which extent parallelism would affect the DGEMM algorithm.