Parallelization of the Matrix Multiply Algorithm

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

***Abstract*—The text discusses findings that result from parallelizing and altering the standard matrix multiply algorithm.**

I. Introduction

This paper will discuss various matrix multiplication algorithms and the performance benefits that may be reaped from different implementation methods.

II. Algorithms

The algorithms employed for the purpose of this study were that of the standard matrix multiply algorithm and the DGEMM matrix multiply algorithm. The latter is a modified version of the former with changes that intend to make better use of cache on the target system.

III. Procedures

The aforementioned algorithms were implemented in the C++ programming language and tested on the XSEDE via the EXPANSE portal. Execution time and gigaflop data was collected within a series of CSV files that were generated by associated runner programs. Each of these programs were run as a singular job on the cluster via SLURM's sbatch command. Lastly, the collected data was analyzed and displayed using MatPlotLib within a Jupyter Notebook running a Python 3 kernel.

IV. Findings

A. The Standard Algorithm

The standard algorithm was implemented as a triply nested for loop used to calculate each element of the target matrix using the provided sources.

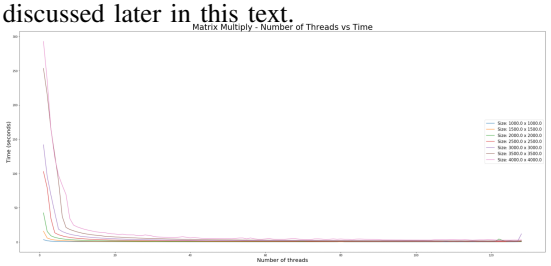
```
void serial_matrix_multiplication(double *A, double *B, double *C, int M, int N, int K) {
    double my_sum;
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            my_sum = 0;
            for (int k = 0; k < K; k++) {
                my_sum += A(i, k, K) * B(k, j, N);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

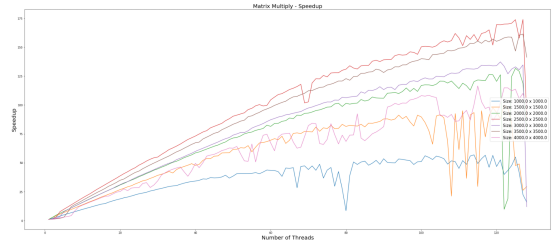
Given that the C++ programming language was employed, a series of pointers are passed into the defined function where each of which references the starting address of an array in memory. The other parameters passed in are used to denote the dimensions of each referenced matrix.

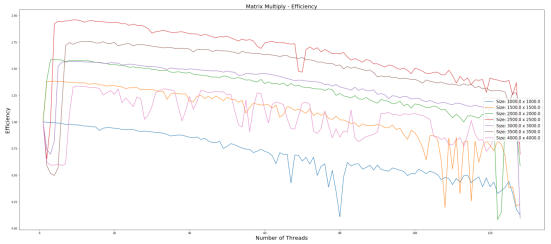
The same algorithm shown above was also parallelized using the OpenMP library by adding a singular pragma, or compiler directive, to the code. This addition allowed the same algorithm to run on more than one thread at the same time which then allows it to take advantage of parallel resources, should they be available on the target system.

```
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;
        }
    }
}
```

The image above illustrates the parallel implementation using the OpenMP library. The results of testing the first algorithm along with parallelizing it showed that processing speeds could be substantially improved by allowing the operations to run at the same time. However, the findings also illustrated that there are limits to this benefit which are imposed on the algorithm by the system and its associated hardware. In viewing the images below, one may ascertain that regardless of matrix size, as the number of threads increases a performance wall will eventually be hit. The only way to circumvent this wall is either to run the code on a better system, or change the algorithm to illicit better performance. This is a concept seen in the implementation of the DGEMM algorithm which is discussed later in this text.







B. The DGEMM Algorithm

```
// initial code provided by assignment handout via Dr. Jones
void do_block(int n, int si, int sj, int sk, double *A, double
*B, double *C, int block_size) {
    for (int i = si; i < si + block_size; ++i)
        for (int j = sj; j < sj + block_size; ++j) {
            double cij = C[i + j * n];/* cij = C[i][j] */
            for (int k = sk; k < sk + block_size; k++)
                cij += A[i + k * n] * B[k + j * n];/* cij+=A[i][k]*B[k][j] */
            C[i + j * n] = cij;/* C[i][j] = cij */
        }
}


// initial code provided by assignment handout via Dr. Jones
void dgemm(int n, double *A, double *B, double *C, int block_size) {
    for (int sj = 0; sj < n; sj += block_size)
        for (int si = 0; si < n; si += block_size)
            for (int sk = 0; sk < n; sk += block_size)
                do_block(n, si, sj, sk, A, B, C, block_size);
}
```

The DGEMM algorithm is an enhancement of the standard matrix multiplication algorithm that places a focus on making better use of cache resources. This is achieved by targeting block sizes, which must be provided to the program at run time. The runner program used to collect data made use of a 50 element block size. The results can be seen in the diagrams below.