OpenMP Matrix Multiplication

Mike Brice

High Performance Computing
Lab 2

Department of Computer Science
Central Washington University
May 2018

Contents

1 Introduction

The purpose of this lab is to perform matrix multiplication to a power and asses the running complexity using different sized matrices and different powers; as well as with different time functions. On top of that this experiment uses openMP to perform the experiments in parallel. OpenMP (open multiprocessing) is a shared memory model parallel programming that takes advantage of multiprocessor computers[4]. OpenMP allows the use of multiple threads running in parallel on a single computer. These threads are implemented to allow each thread to compute one column of the matrix, one row of the matrix, and each element of the matrix. This is possible because each row, column, and element of the matrix do not depend on each other. This lab explores three different methods of computing matrix multiplication in parallel and helps explain which method is more efficient.

2 Methods

2.1 Mathematics

The mathematics behind this experiment are as follows:

$$\begin{gathered}\mathbf{A}^n = \mathbf{A}^{n-1} * \mathbf{A} \\ \mathbf{A}^{n-1} = \mathbf{A}^{n-2} * \mathbf{A} \\ \cdots \\ \mathbf{A}^3 = \mathbf{A}^2 * \mathbf{A} \\ \mathbf{A}^2 = \mathbf{A} * \mathbf{A}\end{gathered}$$

Where A is a m x m square matrix and n is an integer 2,3,4,...,n.

2.2 Computer Specifications

The computer used for these experiments contains the following specifications. An AMD Ryzen 7 1800x 8 core 16 threads 3.6 GHz CPU, 16 GB of RAM, and Ubuntu 16.04 Linux.

2.3 OpenMP Matrix Multiplication

One form of matrix multiplication using openMP is row wise. Each thread gets to compute one row of the result matrix. The pseudo code for this can be seen in Algorithm 1. By using the parallel for loop in openMP, the outer loop can be parallelized. The private command has to be used to insure that each thread gets its own version of the two inner loops. If the private command is not used then k would iterate with each thread, resulting in threads not computing all of the elements of the result matrix [4]. The result of this parallel for loop is that there are m threads each computing its own row with an efficiency of O(m^2).

Algorithm 1 OpenMP Row Wise Matrix Multiplication

```
1: #pragma omp parallel for private(j,k) num_threads(m)
2: for i = 0 to m do
3:     for j = 0 to m do
4:         result[i][j] = 0
5:         for k = 0 to m do
6:             result[i][j] = matrix2[i][k] * matrix1[k][j]
```

Where m is the size of the matrix.

Another form of matrix multiplication using openMP is column wise. Each thread gets to compute one column of the result matrix. The pseudo code for this can be seen in Algorithm 2. By using the parallel for loop in openMP, the outer loop can be parallelized. The second loop from Algorithm 1 was swapped with the outer loop, or loop i and loop j are swapped, in Algorithm 2. This is to remove overhead caused by running a for loop with a parallel for loop nested inside of it [4]. This is also possible because looping row i and looping through column j or looping column j and looping through row i results in the same outcome. The private command has to be used to insure that each thread gets its own version of the two inner loops. The result of this parallel for loop is that there are m threads each computing its own column with an efficiency of $O(m^2)$.

Algorithm 2 OpenMP Column Wise Matrix Multiplication

```
1: #pragma omp parallel for private(i,k) num_threads(m)
2: for j = 0 to m do
3:     for i = 0 to m do
4:         result[i][j] = 0
5:         for k = 0 to m do
6:             result[i][j] = matrix2[i][k] * matrix1[k][j]
```

Where m is the size of the matrix.

The third form of matrix multiplication using openMP is element wise. Each thread gets to compute a single element of the result matrix. The pseudo code for this can be seen in Algorithm 3. This algorithm uses two parallel for loops, the first set of m threads each spawn m threads. This results in m^2 threads or one for each element. Each loop uses the private command to ensure that each thread gets its own k loop. The result of these parallel for loops is that there are m^2 threads each computing its own element with an efficiency of $O(m)$.

Algorithm 3 OpenMP Element Wise Matrix Multiplication

```
1: #pragma omp parallel for private(k) num_threads(m)
2: for i = 0 to m do
3:     #pragma omp parallel for private(k) num_threads(m)
4:     for j = 0 to m do
5:         result[i][j] = 0
6:         for k = 0 to m do
7:             result[i][j] = matrix2[i][k] * matrix1[k][j]
```

Where m is the size of the matrix.

Each Algorithm records its data in a file for use in making plots, such as those found in results. It is important to note that the time efficiencies presented here in the method section do not take into account the time taken by the fork join operations. To see the pseudo code for matrix multiplication to a power, see Algorithm 5 in Appendix B.

The linux command used to compile the code is:

gcc -fopenmp -o Lab2.exe Lab2.c lab2_functions.c -lm

Where -fopenmp is used to tell the compiler to link openMP and -lm is used to tell the compiler to link the math libraries. The math libraries are used in checking user input. On some machines there needs to be a -std=c99 added to the end of the command to compile the code, which tells the compiler to use the c99 standards.

3 Results

3.1 Row Wise

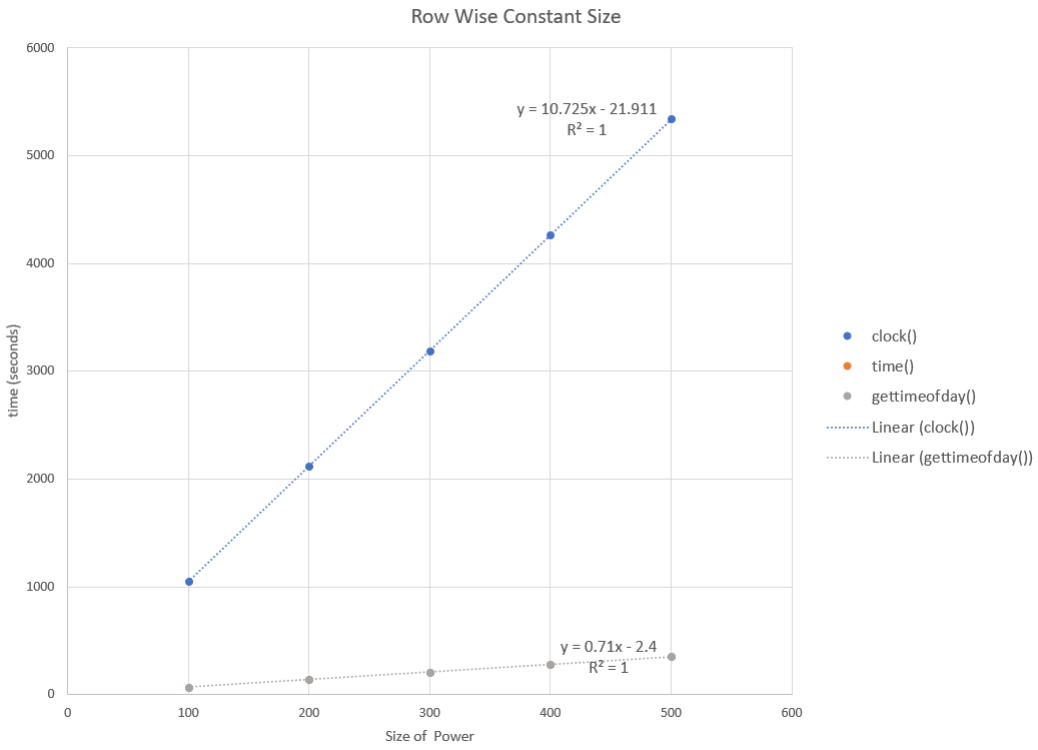

Figure 1: Row Wise Constant Size comparing clock(), time(), and gettimeofday().

It can be seen here that clock() is not the best function to measure the time of a parallel algorithm. This is because clock() measures the combined CPU time of each thread [1]. A better function would be *omp_get_wtime*(); which measures the wall time for the entire parallel block. The constant size produces a linear trend, which would be expected because the O is O($n * m^3$) and since m^3 is held constant, therefore what is left is 0(n). The time is less than the times produces sequentially, which is compared from the previous lab [2].

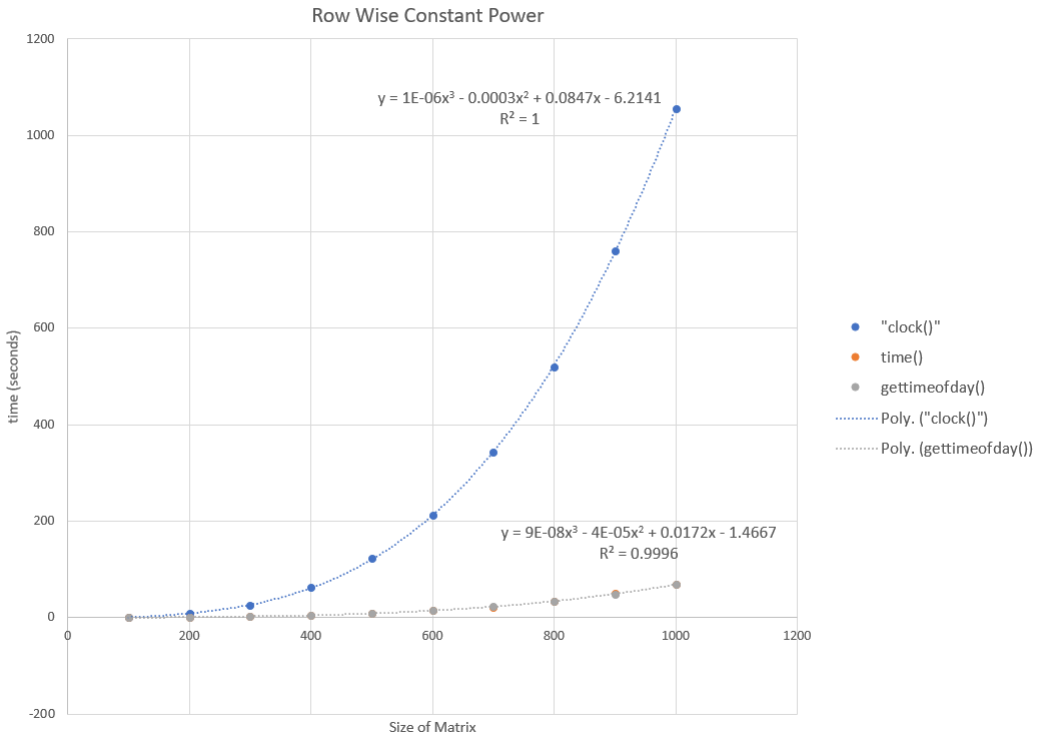

Figure 2: Row Wise Constant Power comparing clock(), time(), and gettimeofday().

It can be seen here that clock() is not the best function to measure the time of a parallel algorithm. The time efficiency is still cubic with constant power, but this was expected because the algorithm had to compute m threads each running m^2 efficiencies. However, the time is less than the times produces sequentially, which is compared from the previous lab [2].

3.2 Column Wise

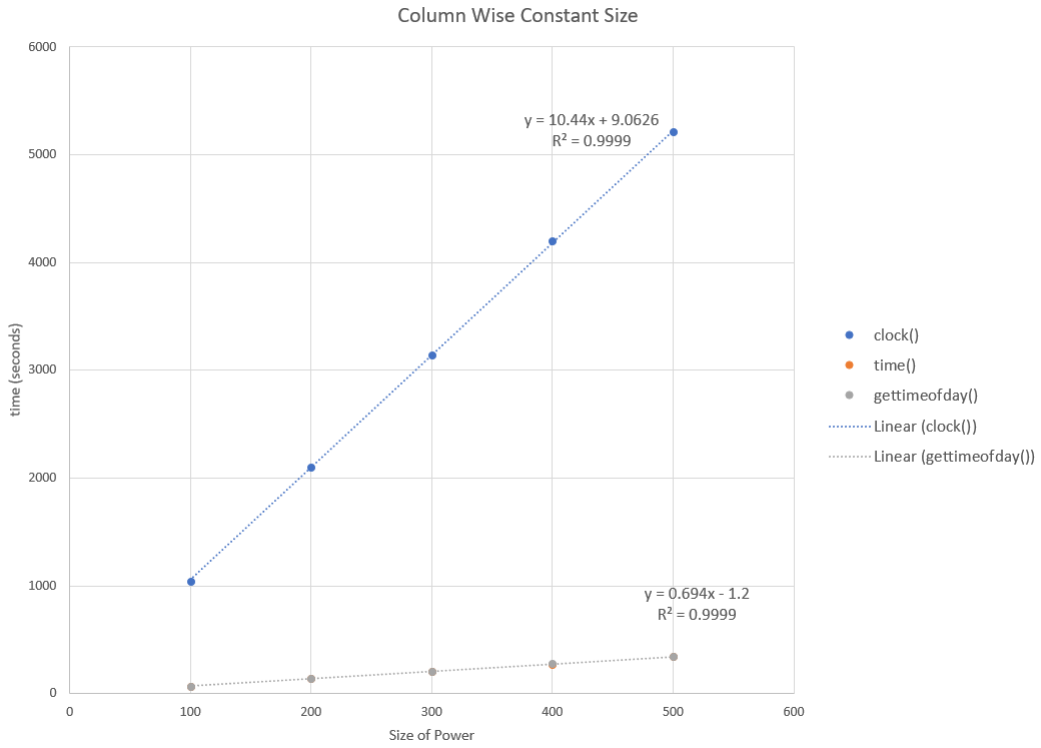

Figure 3: Column Wise Constant Size comparing clock(), time(), and gettimeofday().

Aside from clock(), this result is expected due to O($n * m^3$) and since m^3 is held constant, therefore what is left is 0(n). It is also good to notice that Column Wise and Row Wise have very similar plots and trends. The time is less than the times produces sequentially, which is compared from the previous lab [2].

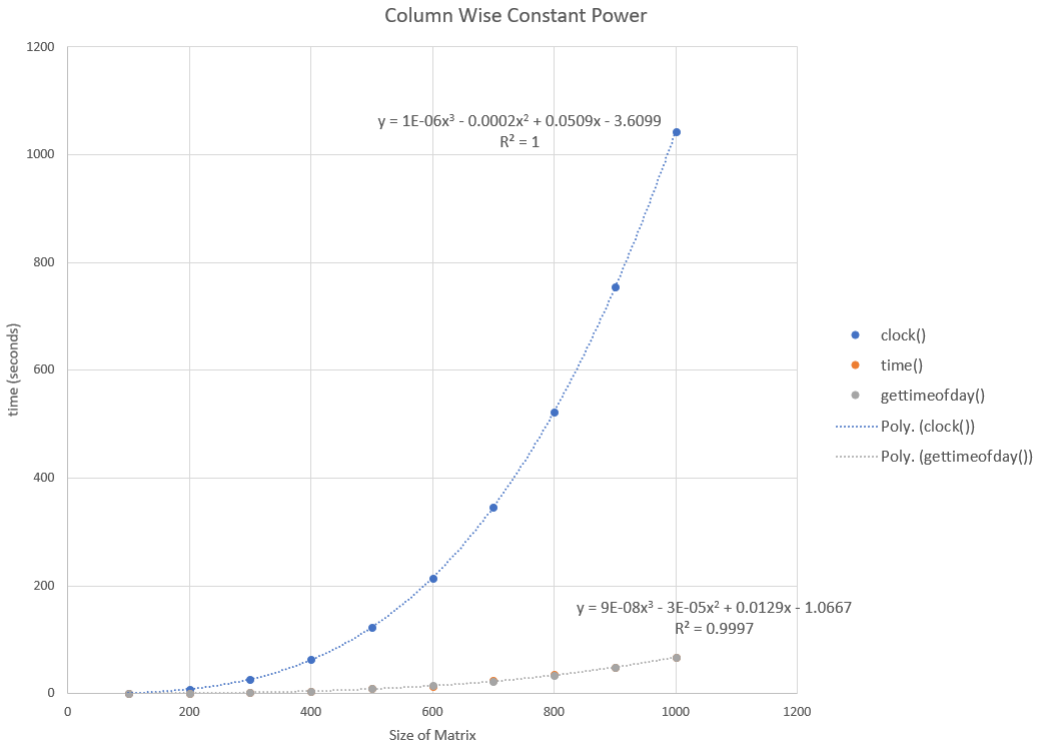

Figure 4: Column Wise Constant Power comparing clock(), time(), and gettimeofday().

Aside from the clock() function, this result is also expected. It is also good to notice that Column Wise and Row Wise have very similar plots and trends. The time is less than the times produces sequentially, which is compared from the previous lab [2].

3.3 Element Wise

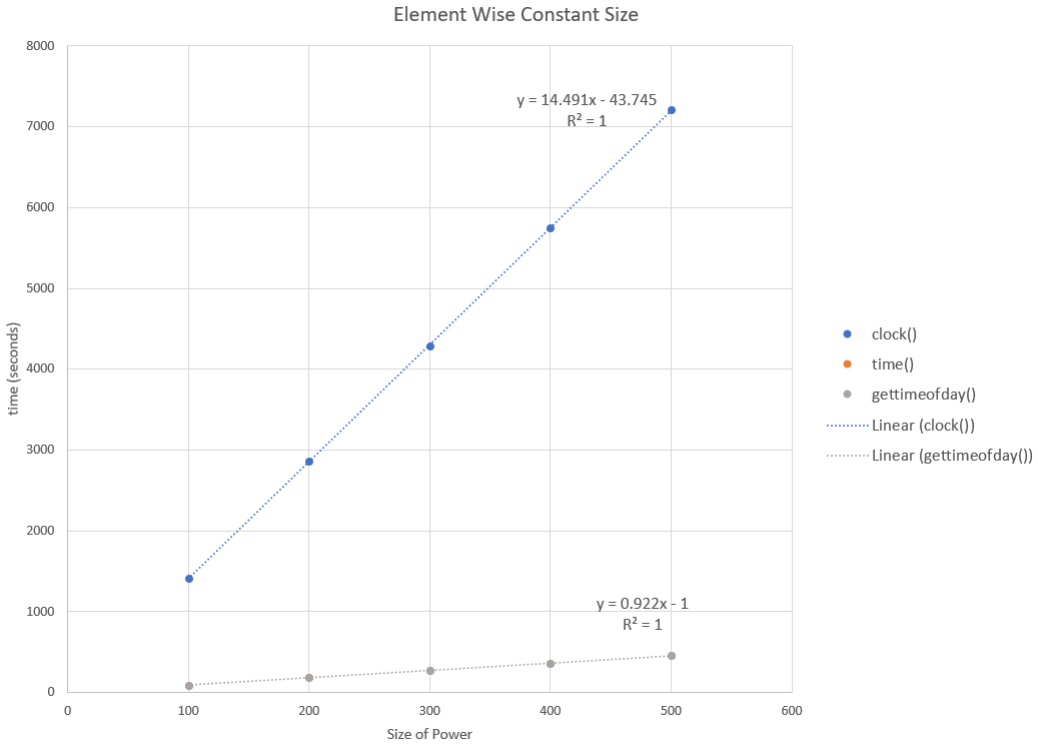

Figure 5: Element Wise Constant Size comparing clock(), time(), and gettimeofday().

Aside from clock(), Element Wise shows expected results. Element Wise when compared to the other methods, which can be seen in the next subsection, is the slower of the three methods. However, the time is still less than the times produced sequentially [2].

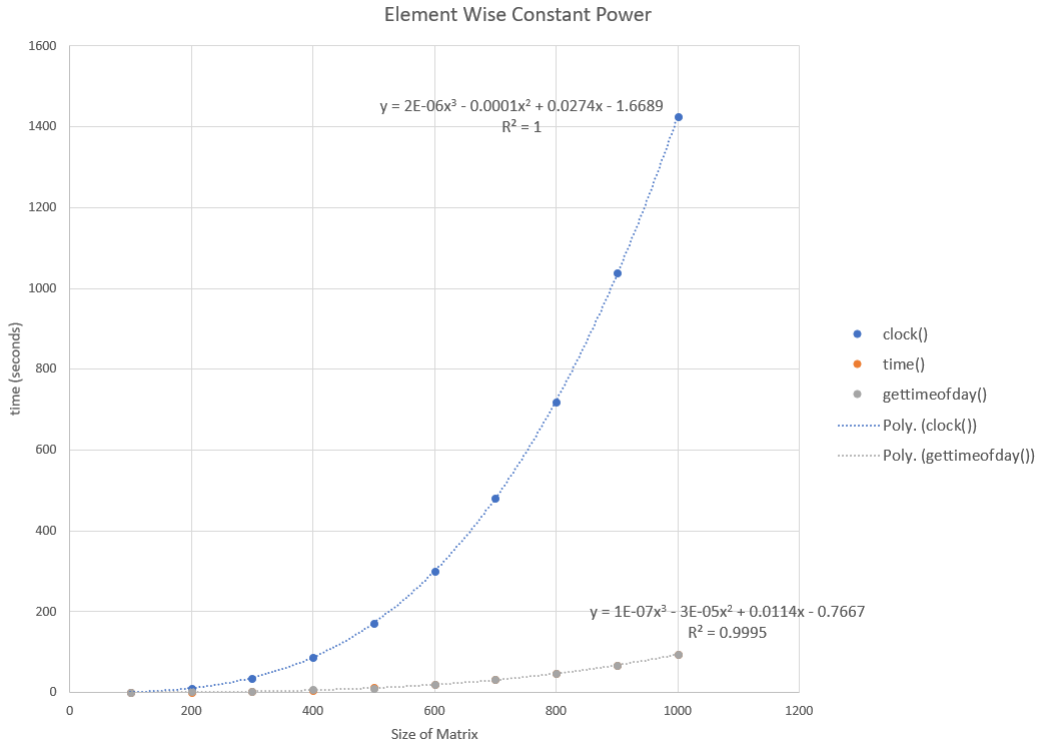

Figure 6: Element Wise Constant Power comparing clock(), time(), and gettimeofday().

3.4 Comparison

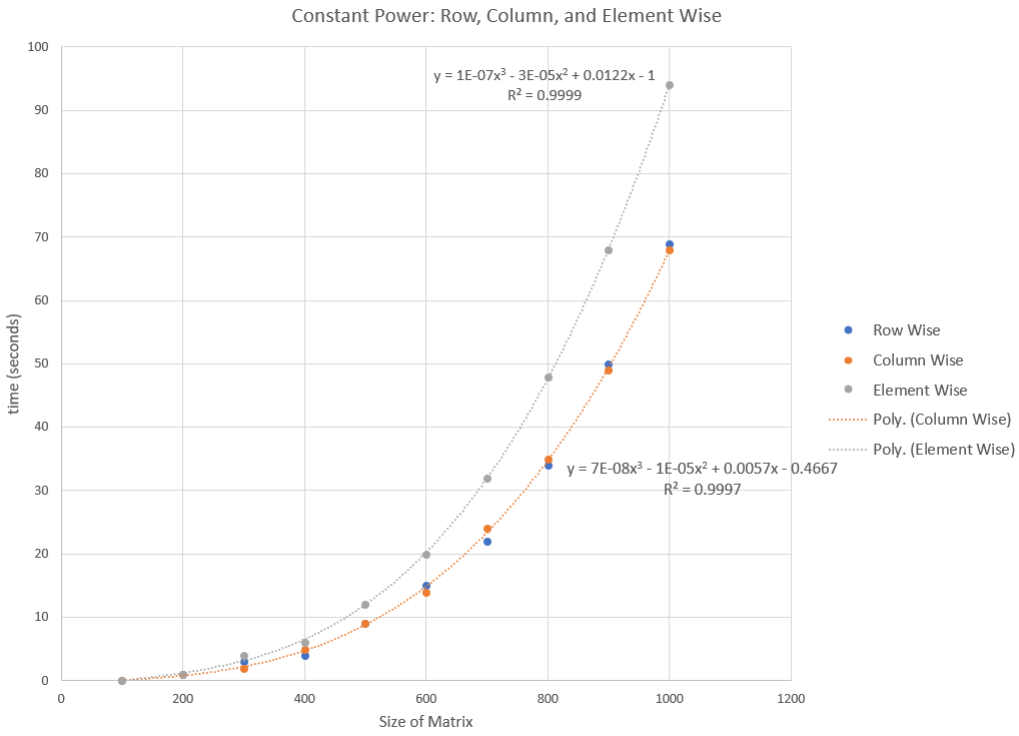

Figure 7: Comparison of Row, Column, and Element Wise using the time() function with constant Power.

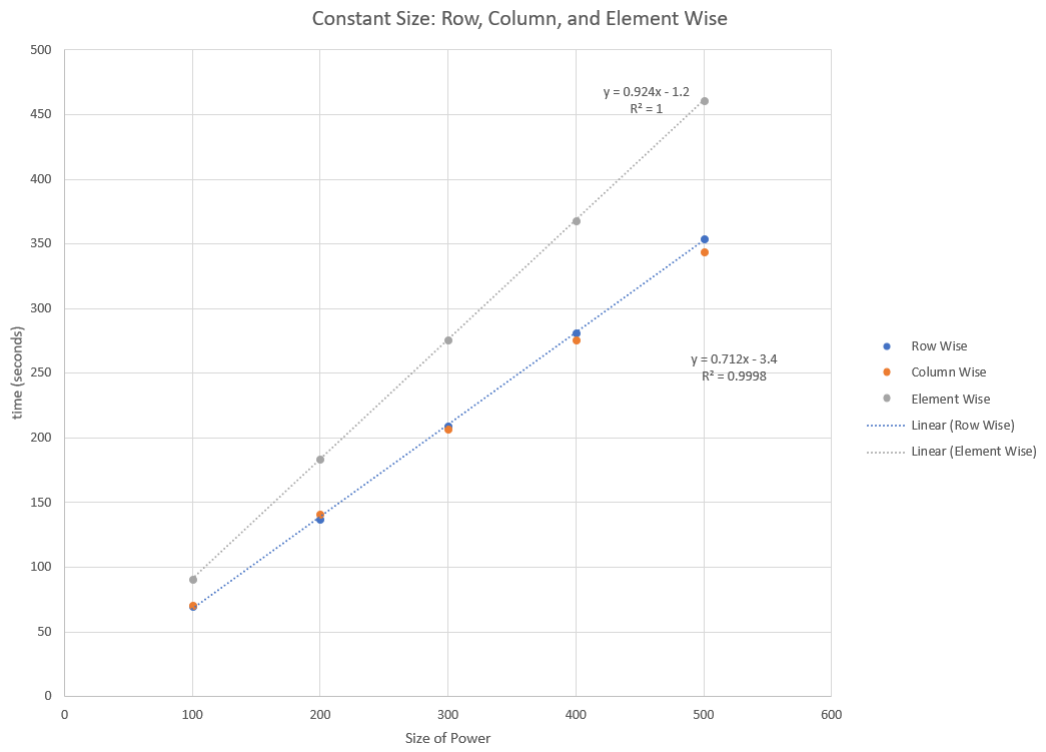

Figure 8: Comparison of Row, Column, and Element Wise using the time() function with constant size.

Based on Figures 7 and 8, Row Wise or Column Wise are the quicker methods over Element Wise, which is expected because Row Wise and Column Wise each produce m threads that run O(m^2) efficiencies where Element Wise produces m^2 threads that run O(m) efficiencies. Since Element Wise has more threads, there are more fork and join operations which produces more overhead. For Element Wise there are m^2 fork and join operations while for Row and Column Wise there are m fork and join operations.

4 Discussion

In this lab, matrix multiplication was explored using openMP in three different methods. These methods at first glance have their advantages and disadvantages. For example, Row Wise and Column Wise both produce m threads but each thread runs at O(m^2) efficiency where as Element Wise produce m^2 threads but each thread runs at O(m) efficiency. The advantage of Row Wise and Column Wise over Element Wise is that they produce less threads, however the disadvantage is that each thread has a slower efficiency than Element

Wise. As evident in section 3.4, Row Wise and Column Wise perform better than Element Wise. This is because the creation of more threads (fork) and the joining of the threads at the end takes a lot of time and Row Wise and Column Wise have less fork/join operations than Element Wise [4]. Even though each thread in Element Wise only computes O(m), the time added by the m^2 fork join operations is greater than the time produced by each Row and Column Wise thread with O(m^2) and m fork join operations.

When comparing to sequential matrix multiplication to a power, each of the parallel methods presented here performs better than their sequential counter part. This was expected due to the fact that each element of the result matrix is independent of each other, which creates excellent conditions for parallelization.

The clock() function is not a well suited function for measuring the time of parallel code. This is because clock() measures the combined time of all the threads while time() and gettimeofday() measure the wall clock [1]. A better suited function for openMP would be omp_get_wtime() which returns the wall clock of the entire parallel block [4].

The method that would have the least amount of overhead would be one that uses p threads with each thread receiving m/p rows or columns in a cyclic pattern, where p is the number of cores / threads on the CPU and m is the size of the matrix. A cyclic pattern is one in which each core gets a new task after it finishes its previous task. The new task is row i + p in the task work flow. Another approach would be to load all of the m/p rows or columns into the thread associated with a core at the beginning and let the cores run and at the end merge the results into one matrix. This method, along with CUDA and MPI could be explored in future experiments. MPI is a Message Passing Interface that can utilize multiprocessors like openMP but can also utilize multicomputers [4]. CUDA is used for writing GPU code.

Appendices

A Sequential Matrix Multiplication

Algorithm 4 Sequential Matrix Multiplication

```
1: for i = 0 to m do
2:     for j = 0 to m do
3:         result[i][j] = 0
4:         for k = 0 to m do
5:             result[i][j] = matrix2[i][k] * matrix1[k][j]
```

Where m is the size of the matrix or A is a mxm matrix. Algorithm 4 is the pseudo code for sequential matrix multiplication. Algorithm 4 is implemented in the previous lab [2]. Sequential matrix multiplication has a time efficiency of $O(m^3)$.

B Sequential Matrix Multiplication to a Power

Algorithm 5 Sequential Matrix Multiplication

```
1: for p = 0 to n do
2:     matrix2 = result
3:     for i = 0 to m do
4:         for j = 0 to m do
5:             result[i][j] = 0
6:             for k = 0 to m do
7:                 result[i][j] = matrix2[i][k] * matrix1[k][j]
```

Where n is the power of the matrix and m is the size of the matrix, or $\mathbf{A}^n$ where A is a mxm matrix. Algorithm 5 is the pseudo code for sequential matrix multiplication to a power. Algorithm 5 is implemented in the previous lab [2]. Sequential matrix multiplication to a power has a time efficiency of $O(n * m^3)$.

References

[1] Openmp time and clock() calculates two different results: Stack overflow. Available at https://stackoverflow.com/questions/10673732/openmp-time-and-clock-calculates-two-different-results.

[2] Mike Brice. Matrix multiplication and sparse matrices. *CS 530: High Performance Computing*, pages 1–9, 2018.

[3] Tutorials Point. C standard library reference tutorial, Jan 2018. Available at https://www.tutorialspoint.com/c_standard_library/index.htm, Note: used to understand the syntax of the different time functions.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OPenMP*. 2007.