PA05: Multithreaded Matrix Multiplication

Jordan Blocher
Raja Singh
Terence Henriod
Dr. Fred Harris
CS615: Parallel Computing
Monday May 5, 2014

Introduction

Shared memory versions of parallel computing are often much faster than message passing paradigms. Because data does not need to be sent over network connections, but can instead be accessed in the usual manner by each running process or thread, the overhead of message passing is all but eliminated. However, shared memory applications can present their own challenges, such as maintaining memory consistency or being able to afford hardware that has enough Processing Units or large enough memory to be useful for the target application.

In this experiment, single process, multi-threaded programs were used in a shared memory environment using up to 8 CPUs. The programs are designed to perform simple matrix multiplication, so many of the issues regarding memory consistency did not need to be addressed, as memory was only read from by multiple threads of execution, and no memory locations were ever written to by different threads; in short, no data race conditions existed.

Theory

Threaded Applications
In traditional operating systems, programs typically run as "heavyweight" processes. Processes have their own instruction sets, data, stacks, and heaps and so forth. A process is relatively isolated from other processes, and in order for processes to share information, usually some kind of message passing paradigm needs to be constructed. This would be the typical structure found in an MPI environment.

However, a heavyweight process is not always necessary for a task, so the overhead needed to spawn a process and facilitate message passing or complex memory sharing schemes is not always as efficient as might be desired. For this reason, we have *threads*. Threads are "lightweight" processes. Threads can have some of their own system resources, but do share much of their parent process' resources as well, including memory, and just in general carry less creation/destruction overhead. These reasons make threads ideal for simple parallelizable tasks.

It should be noted that threads, just like processes, can still only utilize a CPU for processing one at a time. The proportion of CPU time required by a thread affects the degree of multithreading an application can have; threads that do non-CPU dependent activities and need the CPU less allow for the use of more threads per processing core, threads that are very CPU dependent (as in our matrix multiplication) will need the CPU so fewer threads (as few as one per core) is more efficient. The programmer should always keep in mind that the operating system does need to schedule threads along with other threads and processes, so there is no completely deterministic ideal number of threads.

Of course, one should always take their hardware's architecture, operating system, and reservation of cores into consideration, as some hardware or process management schemes facilitate multi-threading very well, while others, like the setup described previously, do not.

Matrix Multiplication

Matrix Multiplication can be summarized as "row times column." This means that each resulting element from a matrix multiplication is found by the summing of pairwise multiplications across the index row of the left side matrix and the index column of the right side of the matrix. Formally defined, each element of a matrix multiplication result matrix is found by:

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} * B_{k,j} \quad [1]$$

where A must be a $p \times n$ matrix and B must be a $n \times q$ matrix. In order for any single resulting element of a matrix multiplication to be computed, all of row *i* of A and all of column *j* of B are required, hence the reason for devising various parallel matrix multiplication algorithms.

A Simple Threaded Matrix Multiplication

An intuitive way to compute matrix multiplication using multiple execution threads is to simply give each thread a share of rows from matrix A to use in computation, use all of the columns of matrix B, and thus compute all of the results of a corresponding set of rows in matrix C. Doing this job allocation in a static manner should be more efficient that attempting a dynamic job allocation scheme. Since all threads will share the same memory as their parent process, there is no delay in terms of memory access and all threads can easily access the matrices as needed.

A Threaded Cannon's Algorithm

Cannon's algorithm utilizes sub matrices in order to reduce the memory usage for any one processor at a time. Sub-matrices are multiplied and then shifted around a mesh composed of torus shaped groups. Cannon's algorithm is easily adapted to use threads in that the same threading method described above can be applied to multiplying the sub-matrices.

(Definitions of speedup and efficiency are listed for completeness, this is not new information)
Speedup

When computing things in parallel, it is advantageous to know how much faster the parallel algorithm/program runs compared to a sequential one. This is one metric used to evaluate the quality of a parallel algorithm, the *speedup factor*. The speedup factor represents how many times faster the parallel algorithm using *p* processors is compared to the sequential one. The speedup factor, *S(p)*, is computed as follows:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{f t_s + (1-f)\frac{t_s}{p}} = \frac{p}{1 + (p-1)f} \quad [2]$$

where *p* is the number of processors used in the parallel algorithm, t_s and t_p are the respective running times for sequential and parallel runs of the algorithms, and *f* is the fraction of the work in a program that must be run sequentially. The second and third expressions are known as Amdahl's law. Ideally, the speedup factor will represent a number of factors $S_p = p$, but this is often not the case due to various factors including non-parallelizable segments of a job or inter-

process communication overhead. If S_p is small relative to p or even approaches 1, then the parallel algorithm should either be improved or not used. Should S_p ever exceed p, this is known as *super-linear speedup*. A super-linear speedup factor is often the product of the use of a runtime that resulted from a sub-optimal sequential algorithm.

Efficiency

Parallel algorithms can also be measured in terms of how well the time to run a program is used by all of the processors. Efficiency measures how well the processors are used (what percentage of the runtime the processors spend working). Efficiency is found by:

$$E = \frac{t_s}{t_p * p} = \frac{S(p)}{p} \quad [3]$$

where t_s, t_p, and $S(p)$ are defined as in [2].

Pseudo-Code

It should be noted that the following pseudo-code should not be considered functional code. Functional code has either been previously submitted or . I have left out many details that would arise in actual code for the sake of brevity and actually describing the algorithms. Each set of pseudo-code assumes that the matrix data has already been read in to the program, so only significant computation and data gathering actions are presented.

Sequential Pseudo-Code

In a sequential program, the entire matrix multiplication procedure is done on one processor, with no additional threads of execution:

Algorithm *MATRIX-MULTIPLY*$(A((1...m)(1...n)), B((1...n)(1...p)))$
$(*$ Input: Two compatibly sized matrices $*)$
1. Matrix C((1...m)(1...p))
2. START-TIMER
3. **for** $i \leftarrow 1$ **to** m
4. **do for** $j \leftarrow 1$ **to** p
5. **do** $c_{i,j} = 0$
6. **for** $k \leftarrow 1$ **to** n
7. **do** $c_{i,j} = c_{i,j} + (a_{i,k} * b_{k,j})$
8. total time = STOP-TIMER
9. **return** C, total time

Parallel Psuedo-Code

In the simplistic version of the parallel algorithm, only one machine ("box") is used, with multiple cores, and multiple threads are spawned to compute various segments of the result matrix C. In this version, all matrix data is made global to ease the burden of passing data to threads:

```
Algorithm THREADED-MATRIX-MULTIPLY(threadCount)
(* Input: The number of threads to be used (matrices A, B, C are global) *)
1.   startRows[], numRowsToDo[] = COMPUTE-SHARES(threadCount)
2.   START-TIMER
3.   for i ←1 to m
4.       do threadID(i) = SPAWN-THREAD(PARTIAL − MATRIX − MULTIPLY,
5.           startRows(i), numRowsToDo(i) )

7.   for i ←1 to m
8.       do THREAD-JOIN(threadID(i))
9.   total time = STOP-TIMER
10.  return total time

Algorithm COMPUTE-SHARES(threadCount)
(* Input: The number of threads that will share the matrix multiplication *)
1.
(* Using integer division on the global A matrix' number of rows *)
2.   baseShare = A.rows / threadCount
3.   remainder = A.rows mod threadCount
4.   for i ←1 to threadCount −1
5.       do startRows(i) = baseShare *(i − 1)
6.           numRowsToDo(i) = baseShare
7.   startRows(threadCount) = baseShare *(i − 1)
8.   numRowsToDo(threadCount) = baseShare + remainder
9.   return startRows, numRowsToDo

Algorithm PARTIAL-MATRIX-MULTIPLY(startRow, stopRow)
(* Input: Indications of which rows in global matrix A to use *)
1.
(* Note that all matrices A, B, and C are global, *)
2.
(* with A being m x n, B being n x p, and C being m x p *)
3.   for i ←startRow to stopRow
4.       do for j ←1 to p
5.              do c_{i,j} = 0
6.                  for k ←1 to n
7.                      do c_{i,j} = c_{i,j} + (a_{i,k} * b_{k,j})
8.
```

Cannon's Algorithm with Threads

```
function MAIN(N) MPI_RANK(rank) MPI_SIZE(numprocs)
    size ← rows/numprocs
    size2 ← size * size
    partitionIndex ← rank * size2 SHIFT(A, size, N, false, true) SHIFT(B, size, N,
true, true) MPI_BARRIER
    for i ∈ [0, blockSize] do MPI_SCATTER(A, A[partitionIndex]) MPI_SCATTER(B,
B[partitionIndex]) MATMULT(A[partitionIndex], B[partitionIndex], C[partitionIndex],
size) MPI_GATHER(C[partitionIndex], C)
        if i != blockSize -1 && rank ==0 then SHIFT(A, size, N, false, false)
SHIFT(B, size, N, true, false)
        end if
    end for
end function
function SHIFT(A, size, N, columnShift, originalAlignment)
    blockSize ← N/size
    size2 ← size * size
    blockSize2 ← blockSize * blockSize
    for row ∈ [0, blockSize] do
        for col ∈ [0, size2] do
            for i ∈ [0, blockSize] do
                if columnShift then
                    r1 ← i * blockSize + row
                    if originalAlignment then
                        r2 ← (i + row) * blockSize%blockSize2 + row
                    else
                        r2 ← (i + 1) * blockSize%blockSize2 + row
                    end if
                else
                    r1 ← row * blockSize + i
                    if originalAlignment then
                        r2 ← row * blockSize + (i + row)%blockSize2
                    else
                        r2 ← row * blockSize + (i + 1)%blockSize2
                    end if
                end if
                A[r1 * size + col] ← A_old[r2 * size2 + col]
            end for
        end for
    end for
end function
```

Results

The use of threads greatly improved performance over the single threaded version. The speedups observed seemed almost unreal. The extreme speedup (super-linear) was likely due to the fact that hardware optimizations for threading made more extremely fast memory referencing, and the fact that shared memory was used meant that no message passing was required; all threads of execution could simply due their computations and "be done with it," so to speak.

It was also observed that using more threads than available cores resulted in minor performance gains. This was likely due to minor multithreading hardware optimizations and the fact that the threads' tasks were primarily computational. Had the threads had more of other kinds of work to do, better performance gains by adding more threads would have likely been observed. That all being said, based on the trend of performance gain by adding more threads, it is likely that using more than 16 threads on 8 cores would not have produces any gain in performance.

Output Data Summaries

Computing Nodes Used in Experiment					
0:	compute-1-32	3:	compute-1-32	6:	compute-1-32
1:	compute-1-32	4:	compute-1-32	7:	compute-1-32
2:	compute-1-32	5:	compute-1-32		
Sequential 0: compute-41-16					

Table 1: The processor nodes used for the basic threaded matrix multiplication program using up to 8 threads.

(More summaries to follow, whitespace is used to accommodate figures and their corresponding tables existing on the same page)

Average Sequential Run Times						
	Matrix Size					
	800	1600	2400	3200	4000	4800
Run Time (seconds)	11.83	110.33	286.33	1842.92	1327.29	5849.56

Table 2: The average running times for the sequential matrix multiplication program

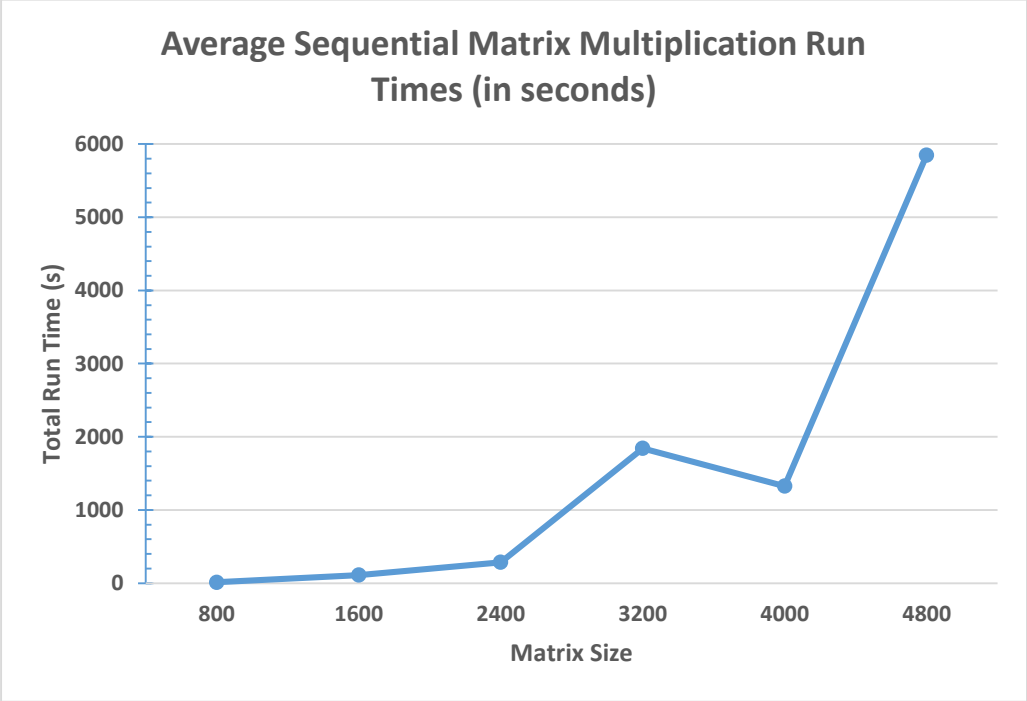

Figure 1: A graph representing the sequential matrix multiplication run times

Average Threaded Matrix Multiplication Run Times with 8 Cores Available (in seconds)							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	2	2.580	25.872	85.670	245.206	396.823	855.287
	4	1.296	13.481	46.458	135.864	217.931	471.820
	6	0.873	9.095	31.923	99.542	149.922	343.234
	8	0.911	8.904	27.935	87.573	136.643	301.377
	10	0.820	7.572	25.246	78.155	121.107	271.037
	12	0.751	6.945	23.141	74.446	111.431	258.513
	14	0.722	6.682	23.233	74.481	118.377	261.646
	16	0.704	6.298	22.596	73.903	115.011	254.126

Table 3: The average running times for matrix multiplication using various numbers of threads distributed across 8 processors

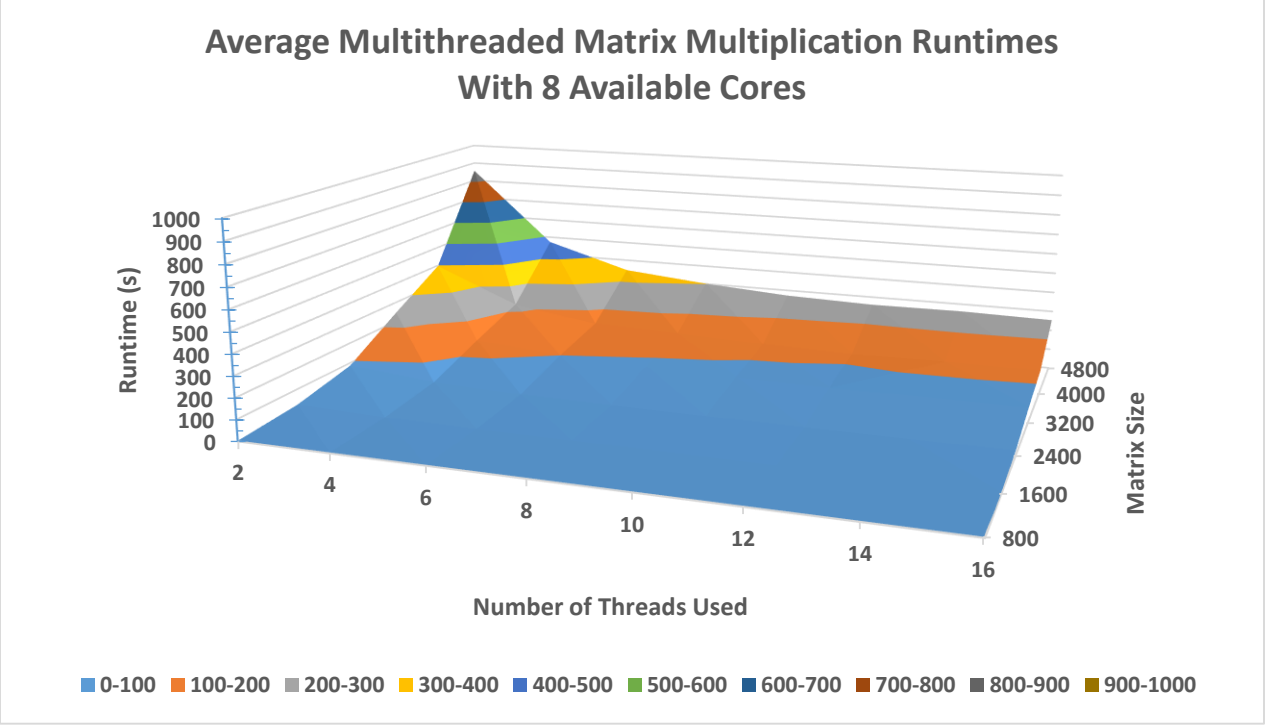

Figure 2: A graph representing the average run times for the multithreaded matrix multiplication program

Average Threaded Matrix Multiplication Speed Up with 8 Cores Available							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	2	4.583	4.264	3.342	7.516	3.345	6.839
	4	9.127	8.184	6.163	13.564	6.090	12.398
	6	13.553	12.130	8.969	18.514	8.853	17.042
	8	12.986	12.391	10.250	21.044	9.714	19.409
	10	14.417	14.569	11.342	23.580	10.960	21.582
	12	15.753	15.886	12.373	24.755	11.911	22.628
	14	16.387	16.512	12.324	24.743	11.212	22.357
	16	16.790	17.516	12.672	24.937	11.541	23.018

Table 4: The speedups observed for matrix multiplication using various numbers of threads distributed across 8 processors

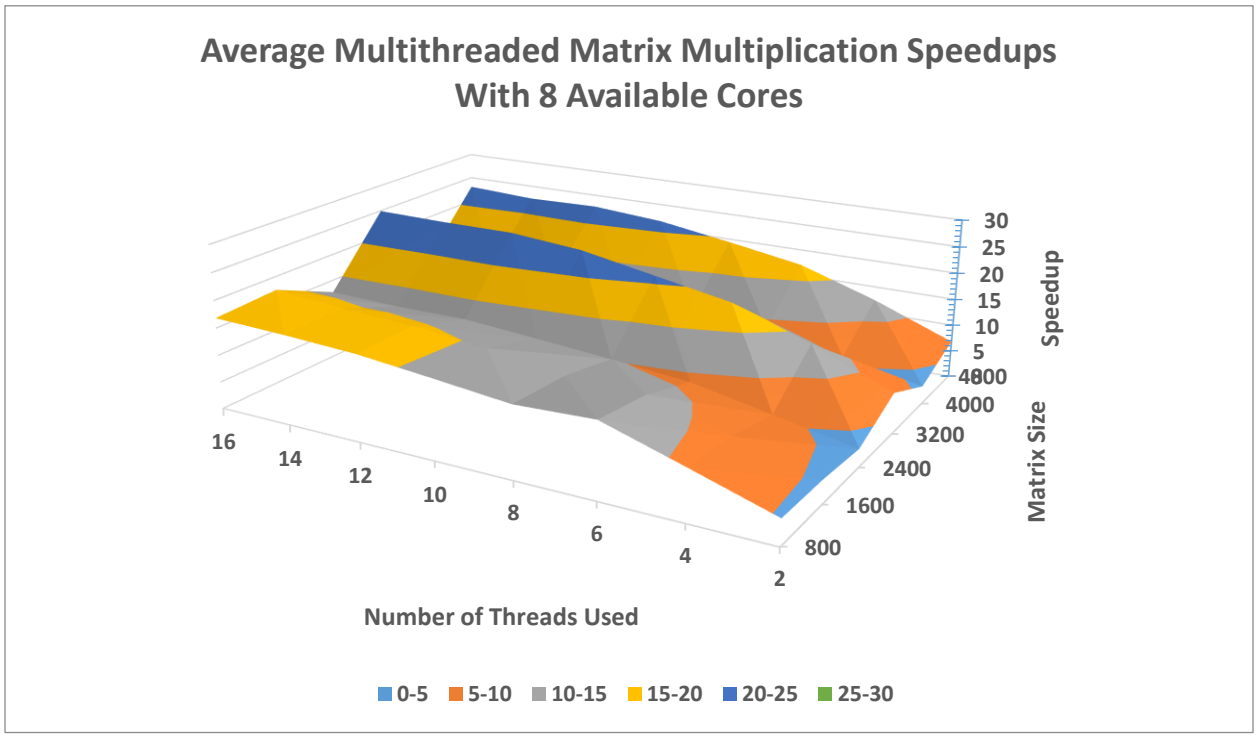

Figure 3: A graph representing the average speedups for the multithreaded matrix multiplication program

Average Threaded Matrix Multiplication Efficiency with 8 Cores Available (in %)							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	2	229.161	213.218	167.112	375.790	167.239	341.965
	4	228.163	204.595	154.082	339.111	152.260	309.946
	6	225.878	202.169	149.489	308.566	147.553	284.041
	8	162.323	154.884	128.122	263.055	121.420	242.618
	10	144.166	145.694	113.416	235.804	109.596	215.822
	12	131.279	132.386	103.112	206.293	99.261	188.564
	14	117.049	117.943	88.031	176.739	80.088	159.691
	16	104.940	109.478	79.197	155.856	72.128	143.865

Table 5: The observed efficiencies relative to the sequential program for matrix multiplication using various numbers of threads distributed across 8 processors

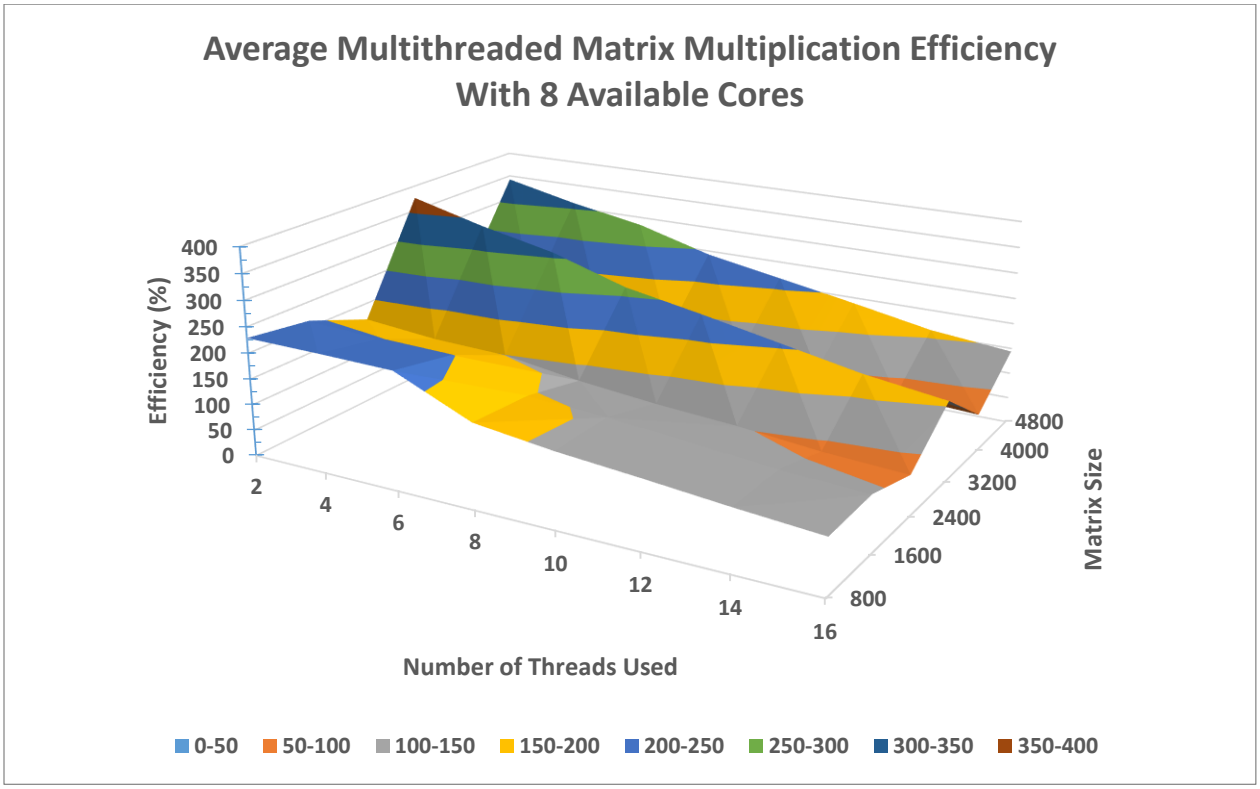

Figure 4: A graph representing the observed efficiencies for the multithreaded matrix multiplication program

Speedup Observed by Doubling the Number of Threads with 8 Cores Available							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads	8 vs 4	2.833	2.906	3.067	2.800	2.904	2.838
	12 vs 6	1.162	1.310	1.380	1.337	1.345	1.328
	16 vs 8	1.293	1.414	1.236	1.185	1.188	1.186

Table 6: An indication of performance gains by using more threads than cores. These "speedups" were computed by simply dividing the runtimes of the trials that used fewer threads by the times of those that used more threads.

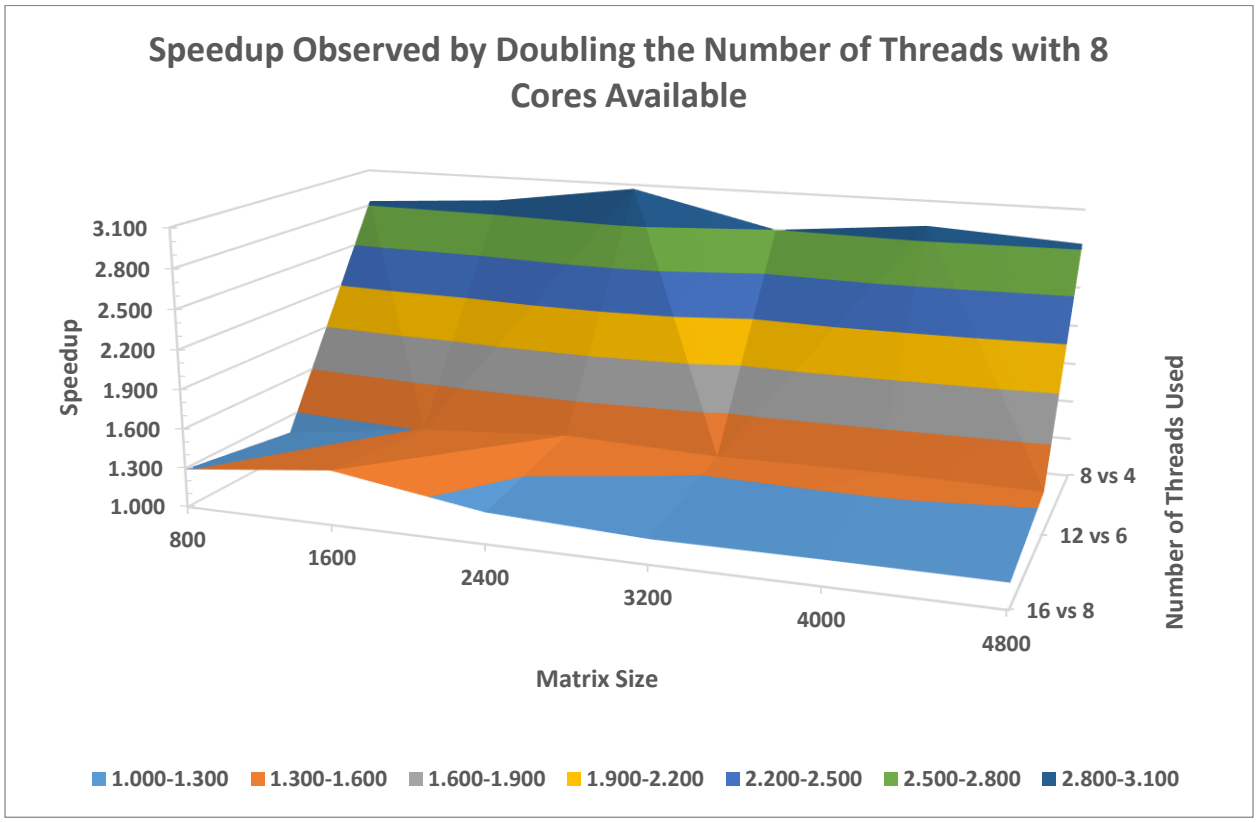

Figure 5: A graph representing the "speedups" gained by using more threads than cores

This Data was lost and is unrecoverable. With the difficulty of getting jobs into the grid, it is not feasible to re-run the trials just to get the node names.

Table 7: The processor nodes used for the basic, non-threaded version of Cannon's algorithm

Average Sequential Run Times						
	Matrix Size					
	800	1600	2400	3200	4000	4800
Run Time (seconds)	18.91	166.03	567.52	1199.30	2383.09	4227.44

Table 8: The running times for the sequential matrix multiplication program used for Cannon's algorithm comparisons

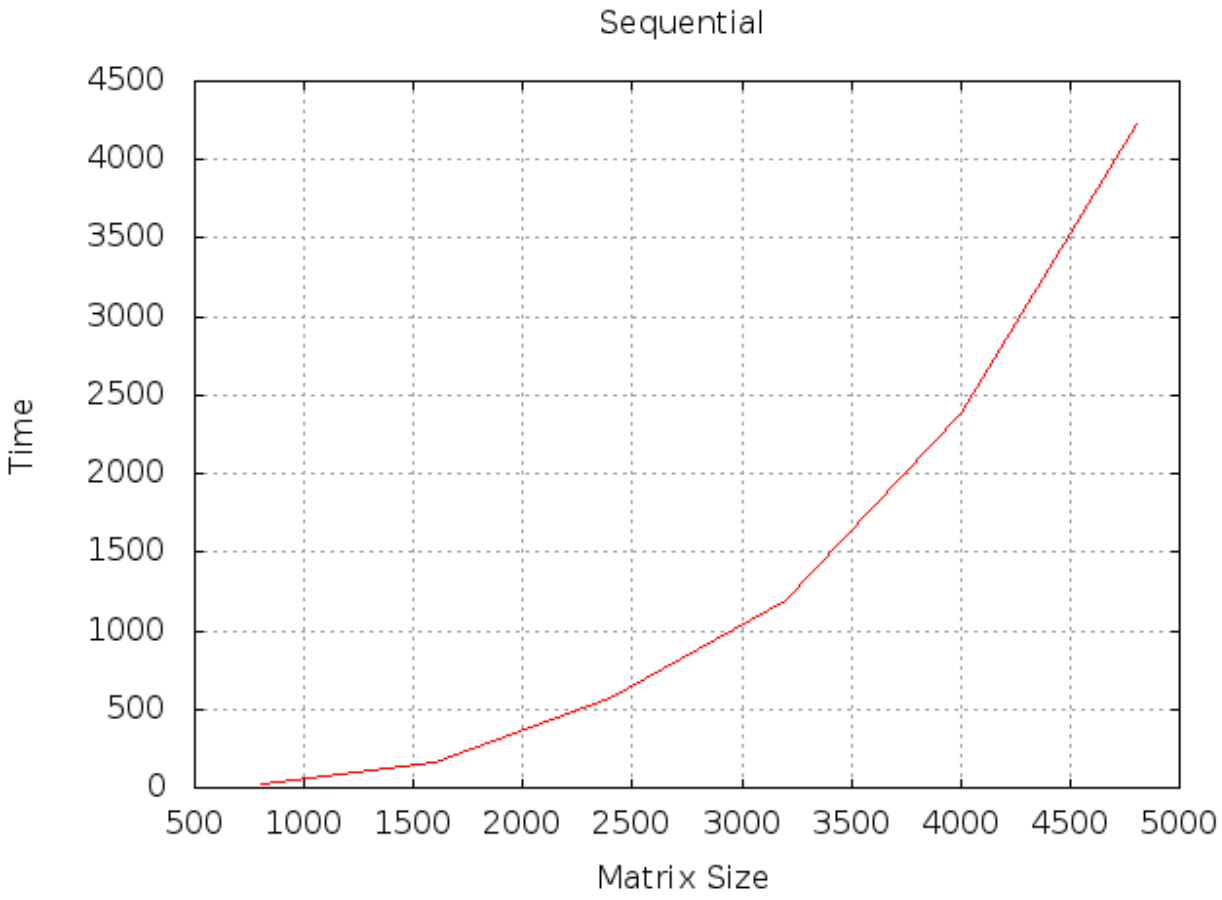

Figure 6: A graph representing the sequential run times used for computing Cannon's algorithm speedups

Runtimes (in seconds) of the Basic Cannon's Algorithm Implementation							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Processors Used	4	8.03	47.96	221.58	531.24	1034.88	1422.14
	16	---	13.30	38.74	61.86	122.41	175.77

Table 9: The running times for the basic, non-threaded version of Cannon's algorithm

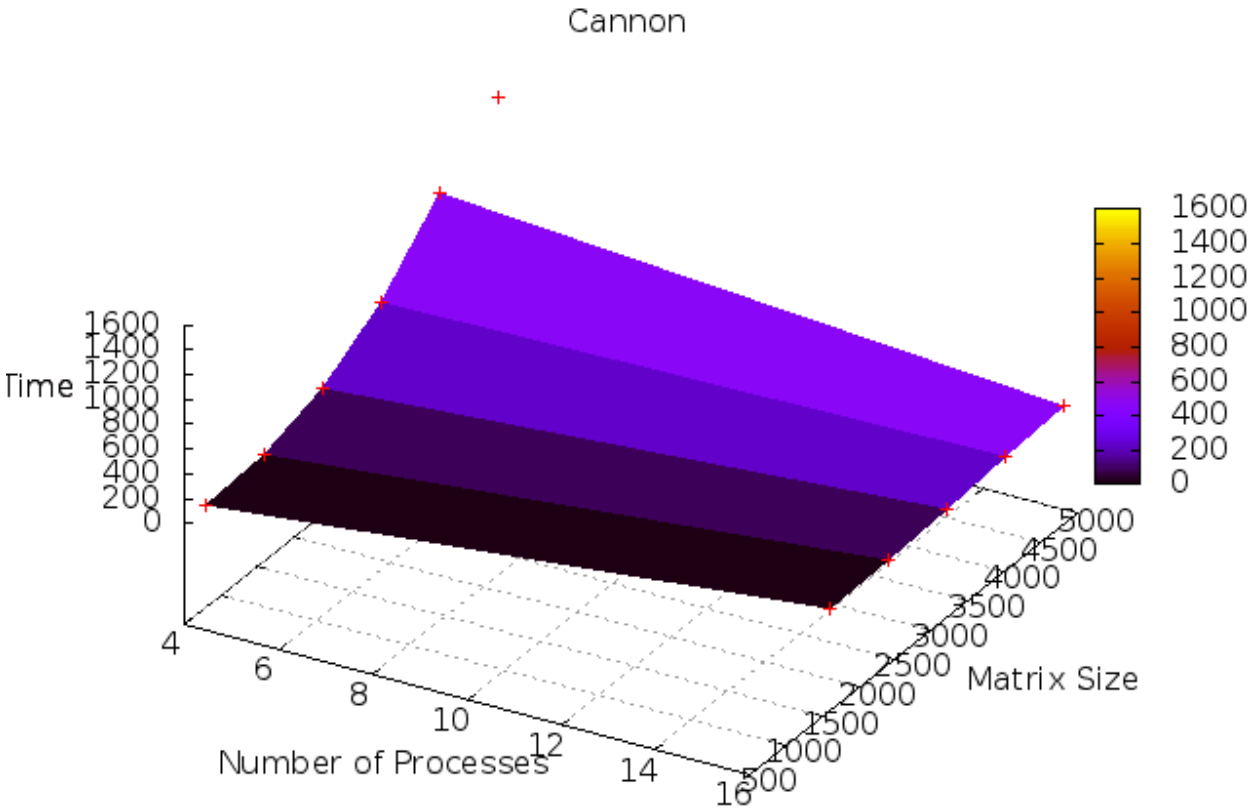

Figure 7: A graph representing the basic, non-threaded version of Cannon's algorithm run times

Runtimes (in seconds) of the Multithreaded Cannon's Algorithm Implementation							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	4	2.739	24.23	77.78	234.74	---	---
	16	1.76	14.73	37.86	81.97	126.76	174.187

Table 10: The average running times for matrix multiplication using the multi-threaded Cannon's algorithm

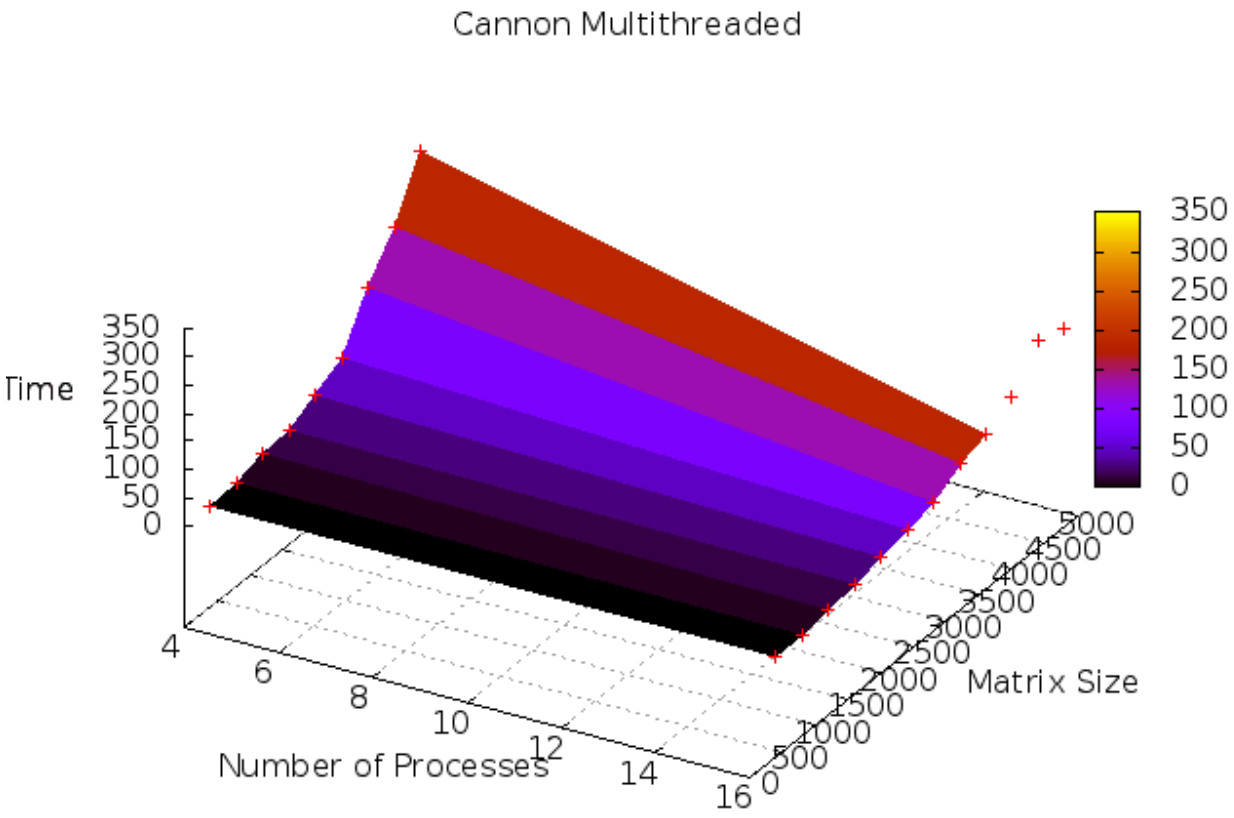

Figure 8: A graph representing the average run times for the multi-threaded Cannon's algorithm

Speedups of the Multithreaded Cannon's Algorithm Implementation							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	4	6.90	6.85	7.30	5.11	---	---
	16	10.77	11.27	14.99	14.63	18.8	24.27

Table 11: The speedups observed for matrix multiplication using the multi-threaded Cannon's algorithm

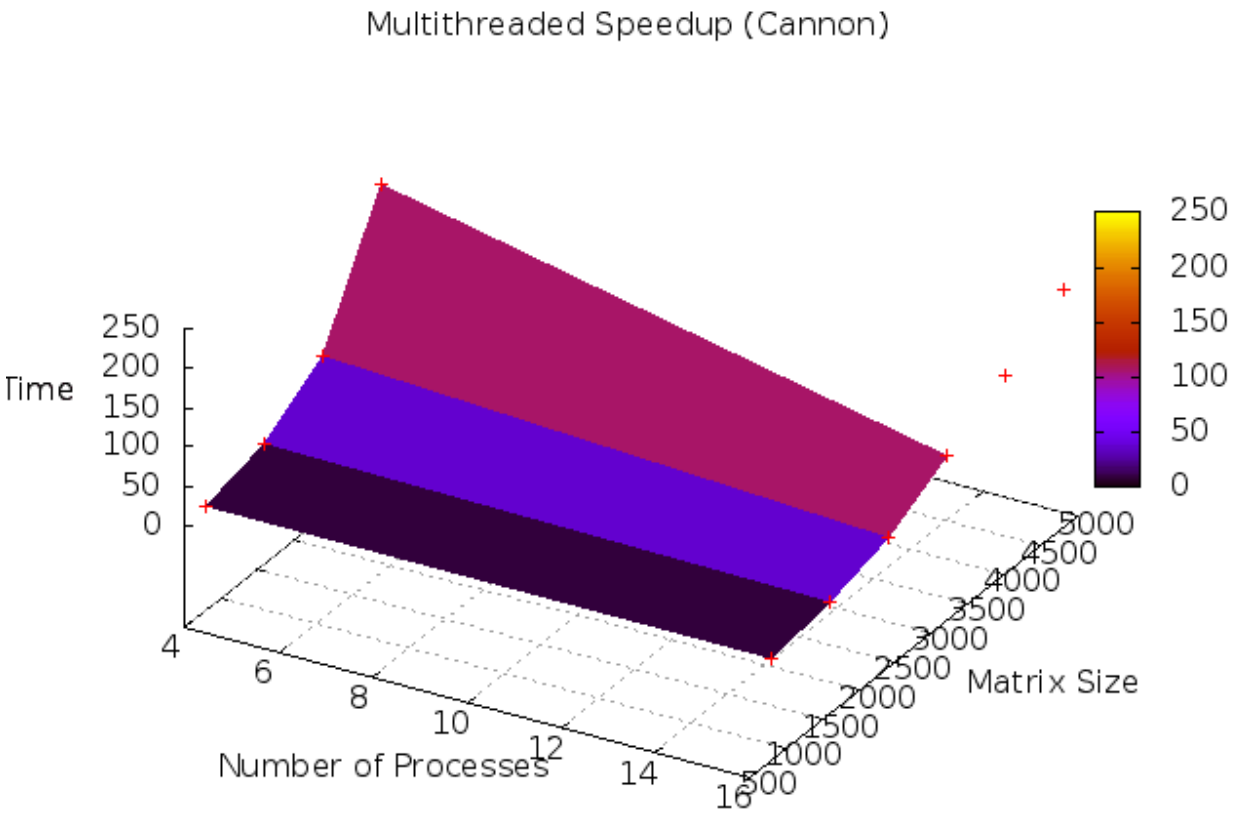

Figure 9: A graph representing the average speedups for the multi-threaded Cannon's algorithm

Efficiency (in %) of the Multithreaded Cannon's Algorithm Implementation							
		Matrix Size					
		800	1600	2400	3200	4000	4800
Number of Threads Used	4	172.62	171.30	182.41	127.72	---	---
	16	67.33	70.45	93.68	91.45	117.51	151.68

Table 12: The observed efficiencies relative to the basic program for the multi-threaded Cannon's algorithm

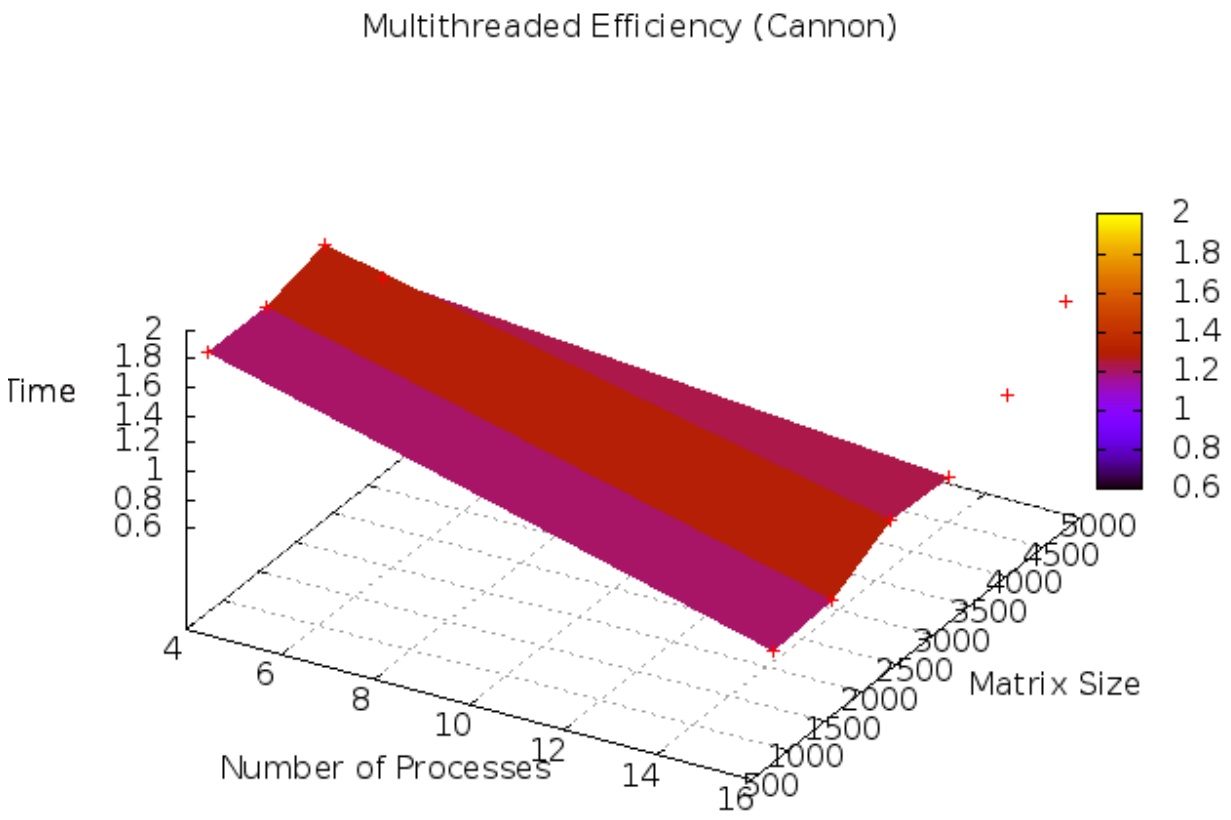

Figure 10: A graph representing the observed efficiencies for the multi-threaded Cannon's algorithm

Issues

We were supposed to try the matrix multiplication using as many as 16 cores, but because someone else was using the machines on the grid that had 16 core machines with shared memory, this did not happen. Also, in order for job simplicity, multi-thread-per-core runs were not done every number of cores possible to use. The rationale behind this was that the performance trends would be apparent without having to trouble ourselves with creating jobs that ran 4 threads on 2 cores and such.