A COMPARIASION OF JOB DURATION UTILIZING HIGH PERFORMANCE COMPUTING ON A DISTRIBUTED GRID

JerNettie Burney, & Robyn Evans, Elizabeth City State University
Mentor: Seung Hee Bae and Jong Youl Choi, Indiana University Bloomington
Principal Investigators: Dr. Geoffrey Fox and Dr. Judy Qui, Indiana University Bloomington

***Abstract*—Parallel computing is defined as carrying out large-scale calculations simultaneously through the use of multiple computing units—such as processors or cores—that work together to devise a solution. During the summer of 2011, undergraduate research students participating in the Science, Technology, Engineering, and Mathematics Initiative at Indiana University Bloomington, were partnered with graduate students to examine the efficiency of parallel computing. To complete this, the team decided to create a program that divides the mathematical computation to multiply large-scale matrices amongst x nodes, or computer cores.**

Overall, the team had a goal of three-fold: i) To understand parallel computing and parallel algorithms for large-scale computing in a shared memory system; ii) To understand the cutting-edge computing technologies needed to maximize the power of multi-core processors; and iii) To learn the standard performance measurement methods behind parallel computing. To these ends, the team implemented parallel matrix multiplication algorithms for a shared memory system and compared their computation performances. The team planned to make comparisons between C, C++, and C#, however, due to time constraints, comparisons were only made between C and C#. These comparisons were made to find not only the most prompt program but also to study the efficiency—the measure of how well the execution was performed—of each one.

A program was first developed to compute sequential matrix multiplication and then was rewritten to include multiple workers to solve the problem; the method for including multiple workers in the program is known as threading. Open Multi-Processing (OpenMP) and Task Parallel Library (TPL) libraries were used in C and C# respectively to specify the number of workers needed to compute the program. The code was then submitted the code to the designated compute node—cn04 for C and cn05 for C#—on the cluster system Storm. It was during this step that both the matrix size and the number of desired workers were specified. The maximum number of threads that could have been used were 24, however 16 was decided to be the current maximum for research purposes. The sequence of threads that were used for calculations were 1, 2, 4, 8, and 16 while the matrix dimensions that were multiplied were 2048 x 2048 and 4096 x 4096. Finally, the time in seconds, it took each job to be completed, was recorded and a comparison was made. (Abstract)

Keywords— cluster, efficiency, node, OpenMP, parallel computing, row decomposition, thread, TPL

I. NATURE AND BACKGROUND OF STUDY

A. Introduction

Matrix computation is one of the many active areas of parallel numerical computing research. Despite this, discussions on parallel matrix-vector multiplication (MVM) algorithms are very rare, often only appearing in a few computing textbooks. MVM problems are simple, yet computationally rich problems, with concepts including domain decomposition and the "communication" between processors. These problems are a wonderful way to explore parallel computing.

This summer, I was amongst a group of students from Elizabeth City State University who traveled to Indiana University's Bloomington (IUB) campus to do just that. Students were partnered with Ph.D. students from the School of Informatics and Computing. The team I was placed on chose to conduct our research in the area of parallel matrix computing with the idea of using the concept to test the efficiency of three popular programming languages: C, C++, and C#.

B. Statement and Background of the Problem

The team's ultimate goal was to examine the efficiency of parallel computing. To answer this question, three objectives were created: 1. to understand parallel computing and parallel algorithms for large-scale computing in a shared memory system; 2. to understand the cutting-edge computing technologies needed to maximize the power of multi-core processors; and 3. to learn the standard performance measurement methods behind parallel computing.

The first step was to create two separate programs—one for each programming language—that multiplied large matrices. To do this, the popular programming languages, C and C#, were utilized. The programs were then rewritten so parallel computing could be utilized. In order to complete this step, both the Open Multi-Processing (OpenMP) and the Task Parallel Library (TPL) libraries had to be accessed. The final step was to then compare the times it took the two programs to run for efficiency.

C. Hypothesis

Due to the team's unfamiliarity with both C and C#, the team hypothesized that there were only three possible outcomes: 1. C would be more efficient than C#; 2. C# would be more efficient than C; or 3. the efficiency for both programs would be nearly the same.

D. Definition of Terms

- **Cluster -** a network of small computers that is under the control of a larger computer
- **Efficiency –** the accomplishment of or ability to accomplish a job with a minimum expenditure of time and effort
- **Job –** a specific task; in this case, the two programs that were created
- **Node –** a core
- **Open Multi-Processing (OpenMP) -** an Application Programming Interface (API) that supports multi-platform shared memory multiprocessing; a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications
- **Parallel Computing –** a form of computing in which many calculations are carried out simultaneously and then solved concurrently ("in parallel")
- **Row Decomposition –** the process of breaking down a task into smaller tasks by row
- **Thread –** a portion of a program that can run independently of and concurrently with other portions of the program
- **Task Parallel Library (TPL) -** a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces in the .NET Framework version 4.; it simplifies the process of adding parallelism and concurrency to applications by vigorously scaling the degree of concurrency to most efficiently use all the processors that are available

II. REVIEW OF LITERATURE

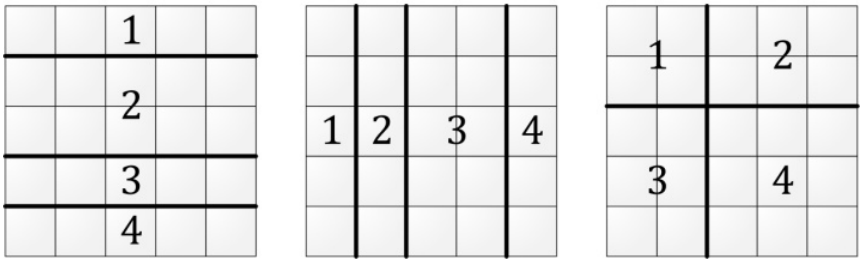
A. Background Research

For data decomposition in parallel MVM, the single steps of calculation that occur in basic matrix multiplication are distributed amongst the available processing units, or nodes.

Depending on the matrix and vector elements, the work is then partitioned over the nodes in a way for the work to be based. In a paper titled, *"Matrix-vector Multiplication in the context of the seminar 'Parallel Programming and Parallel Algorithms'"* Annika Biermann states that the reasoning for this is because, "The more balanced the elements are thereby distributed among the available

processes, the better is the performance and efficiency" of the given results.

There are three simple ways to distribute a MVM to *x* nodes: row-wise, column-wise, and square block-wise. The figure below shows the three different ways to distribute the work load. The figure uses a 5 x 5 matrix that is decomposed amongst four processing units.



row-wise column-wise square block-wise (checkered board)

Figure 1. Examples for different matrix decompositions.

With row-wise decomposition, the matrix is divided into segments and rows of $\left\lceil \frac{m}{p} \right\rceil$, where *m* is the number of rows and *p* is the number of processors. So, in our example, each process is receives either $\left\lceil \frac{m}{p} \right\rceil = \left\lceil \frac{5}{4} \right\rceil = 1$ or $\left\lceil \frac{m}{p} \right\rceil = \left\lceil \frac{5}{4} \right\rceil = 2$ rows of the matrix. Analogously to row-wise decomposition, with column-wise decomposition, each nodes receives $\left\lfloor \frac{n}{p} \right\rfloor$ or $\left\lceil \frac{n}{p} \right\rceil$columns, with *n* representing the number of columns in the matrix and *p* once again representing the number of processors, or nodes.

Now, with the square block-wise decomposition, the matrix is divided into square block dimensions of at least $\left\lfloor \frac{m}{\lceil\sqrt{p}\rceil} \right\rfloor \times \left\lfloor \frac{n}{\lfloor\sqrt{p}\rfloor} \right\rfloor$ or $\left\lfloor \frac{m}{\lfloor\sqrt{p}\rfloor} \right\rfloor \times \left\lfloor \frac{n}{\lceil\sqrt{p}\rceil} \right\rfloor$ and at most $\left\lceil \frac{m}{\lceil\sqrt{p}\rceil} \right\rceil \times \left\lceil \frac{n}{\lfloor\sqrt{p}\rfloor} \right\rceil$ *or* $\left\lceil \frac{m}{\lfloor\sqrt{p}\rfloor} \right\rceil \times \left\lceil \frac{n}{\lceil\lceil\sqrt{p}\rceil\rceil} \right\rceil$. So that means that in our above example, the square blocks' will range in sizes from $\left\lfloor \frac{m}{\lceil\sqrt{p}\rceil} \right\rfloor \times \left\lfloor \frac{n}{\lfloor\sqrt{p}\rfloor} \right\rfloor = \left\lfloor \frac{5}{\lceil\sqrt{4}\rceil} \right\rfloor \times \left\lfloor \frac{5}{\lfloor\sqrt{4}\rfloor} \right\rfloor = 2 \times 2$ to $\left\lceil \frac{m}{\lceil\sqrt{p}\rceil} \right\rceil \times \left\lceil \frac{n}{\lfloor\sqrt{p}\rfloor} \right\rceil = \left\lceil \frac{5}{\lceil\sqrt{4}\rceil} \right\rceil \times \left\lceil \frac{5}{\lfloor\sqrt{4}\rfloor} \right\rceil = 3 \times 3$.

For our research, it was decided to use the row-wise decomposition.

B. Previous Research

Over the past few years, a number of researchers have examined the efficiency of parallel computing and how parallel matrix multiplication can be used to test it. In an article placed in the *Journal of Supercomputing*, Dr. Euncie E. Santos discussed how the effective design of parallel matrix multiplication algorithms relies on the consideration of many interdependent issues. These issues deal with the underlying parallel machine, or the network, upon which the algorithms are to be implemented, as well as the type of methodology utilized by the algorithm. In his research, he determined the parallel complexity of multiplying two necessarily square) matrices on parallel distributed-memory machines and/or networks.

In his research with the Community Grids Laboratory, Ph.D. student, Seung-Hee Bae , designed a parallel SMACOF program using parallel matrix multiplication. For his research, he proposed a block decomposition algorithm based on the number of threads. The proposed block decomposition algorithm worked if the number of row-blocks was at least a half of the number of threads. His paper goes on to further discuss the performance results of the implemented parallel SMACOF in terms of block size, data size, and the number of threads. In addition, a performance comparison was made between a jagged array and a two-dimensional array in C#.

All in all, it was determined that the jagged array data structure performed at least 40% better than the two-dimensional array structure. Bae's research is very similar in concept to the research that was conducted at Indiana University's Bloomington campus for the course of the 2011 summer.

III. METHODOLOGY

A. Definition of the Population

The five recorded trial times, along with the calculated average, for each of the nodes tested acted as the team's test population. All in all, there were five nodes tested—1, 2, 4, 8, and 16— a total of twenty-five duration times recorded, and five averages calculated for both matrices.

B. Procedure

After the decision had been made to only use two programming languages, it was decided that I would work on the C code and that my partner, JerNettie, would work with C#. Due to our lack of experience with the two programing languages, we used online tutorials to become familiar with the two languages.

Once we felt comfortable working with the languages, it was then decided that a visual representation of a generalized mathematical matrix multiplication problem need to be created.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}$$

From here the next step was to recreate the same example using an array format:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} \\ c_{1,0} & c_{1,1} & c_{1,2} \\ c_{2,0} & c_{2,1} & c_{2,2} \end{bmatrix}$$

where the C-matrix is equal to:

$$\begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + a_{0,2}b_{2,1} & a_{0,0}b_{0,2} + a_{0,1}b_{1,2} + a_{0,2}b_{2,2} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} + a_{1,2}b_{2,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,0}b_{0,2} + a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,0}b_{0,0} + a_{2,1}b_{1,0} + a_{2,2}b_{2,0} & a_{2,0}b_{0,1} + a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,0}b_{0,2} + a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}$$

A simple pseudo code was then created and the parameters for the code were determined. The team felt the best way to test the efficiency of the two languages was to used large-scaled matrices in the research; it was then decided that the two test matrices were to be a 2048 x 2048 and a 4096 x 4096.

The next step was to develop program to compute sequential matrix multiplication. Once the programs were successfully created, the team began researching the different types of decomposition that could be used, that is how the program would divide the problem amongst the multiple workers.

There were three direct ways to decompose an *NxN* matrix: row-wise decomposition; column-wise decomposition; and block-wise (or the checkerboard) decomposition. Row-wise decomposition was chosen for this research. The programs were then rewritten to include this threading aspect.

From here, the OpenMP and TPL libraries, within C and C# respectively, were utilized to specify the number of threads. Here is an excerpt of the C code I created:

```
/*
 *  Step 2 : Parallel matrix multiplication
 */

printf("Step 2 : Computing ... \n");

// Set number of threads (workers)
```

```
      omp_set_num_threads(NThread);

// Start time measurement and do computation
      QueryPerformanceFrequency(&timerFrequency);
      QueryPerformanceCounter(&startTimer);

// Put (parallel) matrix multiplication algorithm here
      #pragma omp parallel private(tid)
      {
        tid = omp_get_thread_num();
        printf("Thread %d/%d : Hello\n", tid, NThread);
      }

      QueryPerformanceCounter(&endTimer);
      elapsed = (endTimer.QuadPart - startTimer.QuadPart) * 1000 /
               (double)timerFrequency.QuadPart;

      printf("Elapsed time (millisecond) : %g\n", elapsed);
```

Before the code could be submitted, the 'Remote Desk Log-In' was accessed so we could first log onto the Storm cluster and then into the head node. The code was submitted, via terminal, to the Storm cluster specifying the size of the matrix (either 2048 or 4096) and the desired number of cores (1, 2, 4, 8, or 16) using node `cn-06`. (Note: Originally, the team wanted to use a total of six different cores (1, 2, 4, 8, 16, and 24), but when the code was tested on the 24-thread core, it was decided that execution time was entirely too fast and that it would not be beneficial to include it in our research.) It was here the team encountered a problem. The node we were working on would not grant multiple users access to operate on it at the same time; we had to take turns running our separate codes. This was not acceptable because we were on a tight schedule. The decision was then made to switch to two identical nodes, `storm cn-03` and `storm cn-04`.

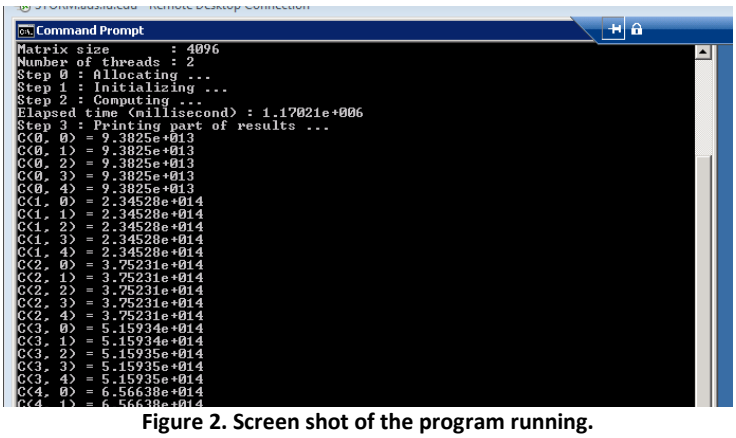


Figure 2. Screen shot of the program running.

For each core, the measure of time, in milliseconds, it took for the program to complete was recorded. This was repeated for a total of five trials the average was

calculated, converted, and recorded in seconds. The efficiency for each core was also calculated and recorded. Our findings were then transferred into a visual representation and comparisons were made.

C. Statistical Methods and Tests that were used to Analyze the Data

Microsoft Excel™ is a commercial spreadsheet application written and distributed by Microsoft for both Microsoft Windows and Mac OS X. It features calculation, graphing tools, pivot tables, and Visual Basic for Applications—a macro programming language. The group specifically used the software to create visual representations of the data collected in the form of charts and graphs. Microsoft Excel™ was also used to calculate both the average time it took the program to run over each core and the efficiency.

IV. ANALYSIS OF DATA

A. Results of the Visual Analysis of Data

After computing the code, the results were as follow:

Matrix Size

2048	**Number of Nodes**				
	1	**2**	**4**	**8**	**16**
Exp #1	176452	88989.9	47987.2	29495.2	12873.7
Exp #2	189759	88240.5	47001.4	28853.7	15376.1
Exp #3	169719	85860.9	49260.7	24510.3	15190.8
Exp #4	200771	94007	55647.5	24595.4	12736.2
Exp #5	196589	96789.1	48710.3	27916.7	12823.3
AVG	186658	90777.5	49721.42	27074.26	13800
Avg. elapsed time (sec)	186.66	90.78	49.72	27.07	13.80
Efficiency	1.00	1.03	0.94	0.86	0.85

Table 1. Times recorded for the 2048 x 2048 matrix.

Matrix Size

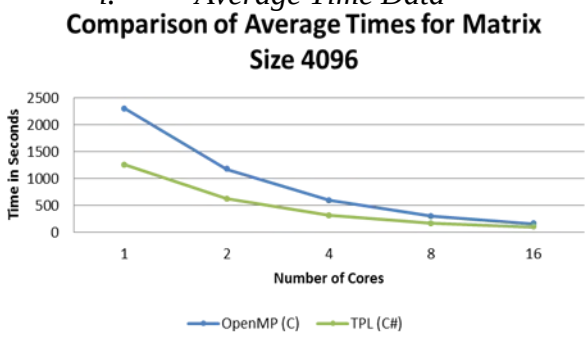
4096	**Number of Nodes**

	1	2	4	8	16
Exp #1	2346700	1171290	579980	306530	145844
Exp #2	2307040	1186250	584977	285423	166713
Exp #3	2253200	1170210	595222	286066	164509
Exp #4	2274620	1224930	597320	299762	149926
Exp #5	2287000	1119270	590686	297686	151074
AVG	2293712	1174390	589637	295093.4	155613.2
Avg. elapsed time (sec)	2293.71	1174.39	589.64	295.09	155.61
Efficiency	1.00	0.98	0.97	0.97	0.92

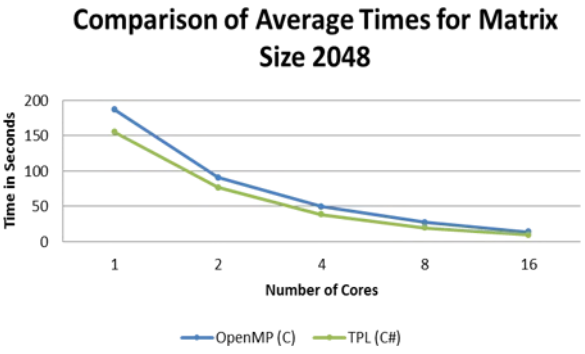
Table 2. Times recorded for the 4096 x 4096 matrix.

D. Tables, figures, etc. used for Data Analysis

i. Average Time Data



Graph 1. Shows the average time it took each program to run using a 4096 x 4096 matrix.



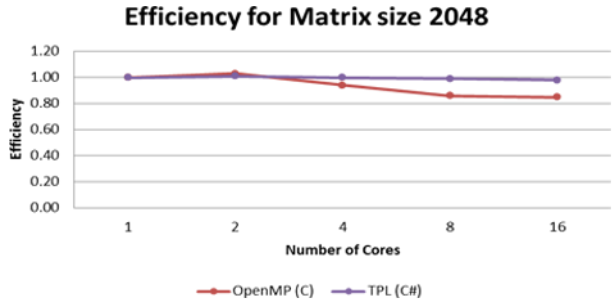
Graph 2. Shows the average time it took each program to run using a 2048 x 2048 matrix.

Average Times

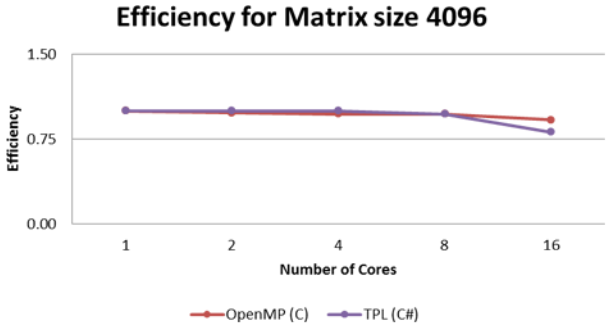
		Number of Cores				
	Matrix Size	1	2	4	8	16
OpenMP (C)	4096	2293.71	1174.39	589.64	295.09	155.61
	2048	186.66	90.78	49.72	27.07	13.80
TPL (C#)	2048	155.22	76.91	38.64	19.57	9.86
	4096	1251.21	622.51	313.85	161.30	96.30

Table 3. Chart used to compare time.

ii. Efficiency Data



Graph 3. Shows the efficiency rate for the 2048 x 2048 matrix program.



Graph 4. Shows the efficiency rate for the 4096 x 4096 matrix program.

Efficiency

		Number of Cores				
	Matrix Size	1	2	4	8	16
OpenMP (C)	2048	1.00	1.03	0.94	0.86	0.85
	4096	1.00	0.98	0.97	0.97	0.92
TPL (C#)	2048	1.00	1.01	1.00	0.99	0.98
	4096	1.00	1.00	1.00	0.97	0.81

Table 4. Chart used to compare the efficiency.

C. Decision about the Hypothesis

The team's hypothesis proved to be correct; that is, that C# was the most efficient language.

V. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

A. Conclusions Resulting from Visual Analysis of the Data

Once the examining the time graphs had been completed, it was seen that OpenMP and TPL nearly had the same finishing times with the 2048 x 2048 matrix. While comparing the 4096 x 4096 matrix, it was noted that the TPL, or C# program, had a much faster completion rate than the OpenMP program. It was then inferred that until a certain size was obtained with the matrices, the time it takes for a job to be completed for both libraries would be very similar, even with an increase in the number of workers. However, the increase in the size of the matrix and the number of workers leads to a huge variance in both libraries—with TPL becoming the least time consuming of the two.

By observing the efficiency for the two libraries, based upon the graph, there is a correlation between the increase of the size of the matrix and number of workers that causes the plots for the graph to constantly change. Though all of the data starts of similarly, it is seen that when the number of workers increases, the effectiveness of the jobs being completed starts to plummet. What is interesting is that with the OpenMP program, unlike the time graph, the efficiency graph went down the most with the 2048 x 2048 matrix rather than the 4096 x 4096 matrix; whereas TPL followed the same suit for efficiency as it did with the time graph, that is that the greater the dimensions and the more workers there were, the less effective it became.

Based upon the observations gathered, it can be concluded that C# works best according to time and effectiveness. However, due to time constraints, the reason for this could not be determined at this time.

B. Shortcomings

Throughout the course of this project, the team faced a few setbacks. Originally the team wanted to examine three different computing languages: C, C++, and C#. However, due to time constraints, only two languages—C and C#—were examined. Originally it was intended for Storm cn06 to be the designated node for testing. But, working on this node would not allow multiple users to operate on the node at the same time; the data collect would have been contaminated. Also, the team had originally planned on using the HPC Job Manager for job submissions, but due to a system error, jobs had to be submitted through terminal.

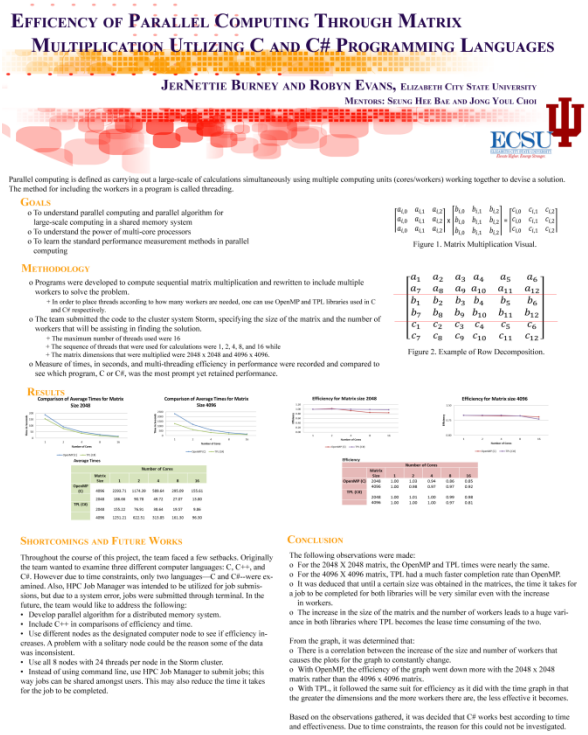
Ultimately, the team's biggest setback was time. There was not enough time to examine why C# was the more efficient of the two programming languages tested. Nor was there enough time to make a conduct the experiment using C++, and then make a comparison between the three.

VI. FUTURE WORKS

Aside from making a comparison between three languages—C, C++, and C#—the team would like to develop a parallel algorithm for a distributed memory system, as well as designate a different node as the "manager" node to see if there will be a change in efficiency—a problem with a solitary node could be the reason for some of the data inconsistency;. Ideally the team would like to use all twenty-four threads within the eight of the nodes in the Storm cluster. Finally, the team would like to find another way to submit the jobs. Instead of using command line to submit the jobs, the team would like to use HPC Job Manager; this would allow resources

to be shared amongst multiple users and could potentially reduce the time it takes for the job to be completed.

VII. Poster Presentation at CIC SROP Conference



In preparation for our trip to the Committee on Institutional Cooperation Summer Research Opportunities Program's (CIC SROP) conference at the Ohio State University, the School of Informatics and computing here at IUB provided us with the opportunity to attend poster presentation workshops. In these workshops, we learned how to prepare a strong poster presentation; specifically, guidelines were given on what to include in the poster (title, affiliations, methodology, results, etc.) and what not to include (references, acknowledgments, etc.).

Taking into consideration what we learned, our team sat down and created two separate designs for our poster. The poster I created is to the above left.

The CIC SROP hosted its 25th conference at the Ohio State University in Columbus, Ohio. Over 400 undergraduate and graduate researchers from the participating universities came together with faculty and staff for a weekend of events. At the conference I presented my research to my peers and attended workshops on receiving funding to graduate school as well as a session on the new and revised graduate record examinations, or the GRE.

ACKNOWLEDGMENTS

Personally, I would like to acknowledge the STEM Initiative Summer Scholars Institute for the opportunity to travel to Indiana University at the Bloomington campus to conduct research for the summer. I would also like to thank IUB's School of Informatics and Computing—including principal investigators Dr. Geoffrey Fox and Dr. Judy Qui; Assistant Dean for Diversity and Education, Dr. Maureen Biggers; Director if Diversity Lamara D. Warren; and my mentors Seung Hee Bae and Jong Youl Choi for the hospitality, encouragement, and devotion the bestowed upon us during our stay.

REFERENCES

Biermann, A. (2010). *Matrix-vector Multiplication in the context of the seminar "Parallel Programming and Parallel Algorithms"*. Westfälische Wilhelms-Universität Münster.

C. Edwards, P. Geng, A. Patra, and R. van de Geijn, *Parallel matrix distributions: have we been doing it all wrong?*, Tech. Report TR-95-40, Dept of Computer Sciences, The University of Texas at Austin, 1995.

Davis, Ian. "Parallel Matrix Multiplication with the Task Parallel Library (TPL)." *Innovation Software*. N.p., 31 Mar 2010. Web

Foster, Ian. "4.6 Case Study: Matrix Multiplication." *Designing and Building Parallel Programs*. N.p., 1995. Web. 12 Jun 2011. <http://www.mcs.anl.gov/~itf/dbpp/text/node45.html>.

Gunnels , John, Calvin Lin, Greg Morow, and Robert van de Geijn . "Analysis of a Class of Parallel Matrix Multiplication Algorithms ." *Analysis of a Class of Parallel Matrix Multiplication Algorithms* . University of Texas, n.d. Web. 14 Jul 2011. <http://www.cs.utexas.edu/users/plapack/papers/ipps98/ipps98.html

"Parallel Matrix-Matrix Multiplication." *Computer Science Department at Indiana University*. Indiana University, 16 Feb 2011. Web. 24 Jul 2011. <http://www.cs.indiana.edu/classes/b673/notes/matrix_mult.html>.

Qazi, Nabeel . "Matrix Vector Multiplication in MPI – Row Wise Block Striped Decomposition." *HPCWatch: High Performance Computing Blog*. 06 Jun 2011. Web. 9 Jul 2011. <http://hpcwatch.com/matrix-vector-multiplication-in-mpi-row-wise-block-striped-decomposition/>.

Santos, Eunice E. "Parallel Complexity of Matrix Multiplication." *Journal of Supercomputing* 25.2 (2003): n. pag. Web. 3 Jul 2011.