Parallelizing Matrix-Vector Multiplication and its Effects on Speedup and Efficiency

Patrick Boronski[1], Rodolfo Croes[2]

[1, 2]Department of Computing Sciences, Coastal Carolina University
Conway, SC, USA

[1]plboronsk@coastal.edu

[2]rjcroes@coastal.edu

***Abstract*— In this work we used pthreads to implement a parallel program that will multiply a matrix by a column vector to see how the time complexity, speedup and efficiency changes for different size matrices and for the amount of threads available.**

***Keywords*— parallel computing, matrix-vector multiplication, speedup, efficiency**

I. Introduction

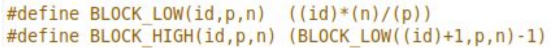
For this project, we were given the challenge of performing matrix-vector multiplication using pthreads with two input matrices read in from files, and the product of said multiplication written out to a file. This code then needed to be tested on COMET in order to measure the execution times, speedup, and efficiency of said program with matrices of three different sizes on different numbers of threads. Before working on the parallel version of this project, we wrote a serial version of the code to test whether it computed the matrix multiplication correctly. The requirements for matrix multiplication are that the number of columns in the first matrix matches the number of rows in the second matrix, and the dimensions of the output matrix is the number of rows from the first matrix and the number of columns from the second matrix. The matrices were made by giving a program the dimensions of said matrix, and filling it with randomized numbers between 0 and 1. The time collection was divided into three data sets: the time spent overall, the time spent only on the computation, and the time spent only on I/O. The computation time and the overall time are the only ones used for plotting the speedup and efficiency chart, because the time spent on I/O has no influence on the performance of the multiplication. However, the I/O time is used for the overall time, as it accounts for file reading/writing.

II. The Code

We began with code written by Dr. Robert Robey, co-author of Parallel and High Performance Computing, which generated a matrix and a column vector, both filled with random data, and multiplied them in parallel using pthreads[1]. The author's code assumed that the number of rows was divisible by the number of threads. Our code allows the rows to be divided unevenly between any number of threads, allowing for more flexibility when running the code on larger data sets or systems with different specifications. We also altered the code so that it reads the matrix and vector from files instead of generating them randomly, allowing for pre-generated files to be used by the program. The code that we used to generate the matrices fills them with random data and writes them to a file, but this could easily be swapped out with other code so long as the output format remains the same.

A. Row Division

To calculate which threads will receive which rows, we used macros written by Michael J. Quinn, author of Parallel Programming in C with MPI and OpenMP[2].

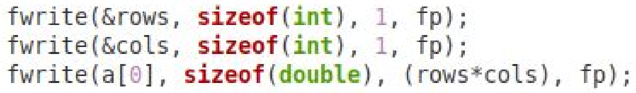


```
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
```

These macros calculate the starting and ending indices, inclusive, of a thread given that thread's rank, the total number of threads, and the number of rows in the matrix.

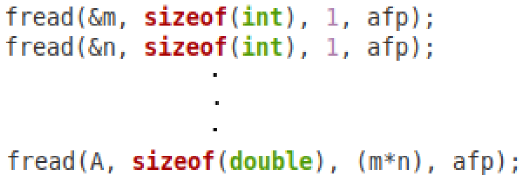
B. File Format

To read and write the data in the matrices to and from the files, we wrote the data in binary. The data is written to and read from the file in the order <rows><columns><data> requiring three separate reads or writes.



```
fwrite(&rows, sizeof(int), 1, fp);
fwrite(&cols, sizeof(int), 1, fp);
fwrite(a[0], sizeof(double), (rows*cols), fp);
```

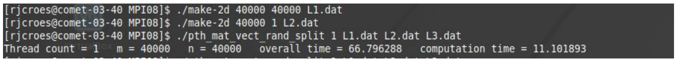
So long as this order is kept, any code that generates a matrix can be used in place of our matrix generation code using the following reads.



```
fread(&m, sizeof(int), 1, afp);
fread(&n, sizeof(int), 1, afp);
              .
              .
              .
fread(A, sizeof(double), (m*n), afp);
```

III. Timing Data

Once we had code that could run on any number of threads, with many different sizes of matrices, we collected data for the overall execution time, the time spent doing only the computation, and the time spent on file I/O. All data was collected on one node of the supercomputer COMET. Because each COMET node has 24 cores, we collected data for thread counts of up to 24. We collected data on three different sized matrices, up to 40k x 40k. I/O time was calculated by subtracting computation time from total time.



```
[rjcroes@comet-03-40 MPI08]$ ./make-2d 40000 40000 L1.dat
[rjcroes@comet-03-40 MPI08]$ ./make-2d 40000 1 L2.dat
[rjcroes@comet-03-40 MPI08]$ ./pth_mat_vect_rand_split 1 L1.dat L2.dat L3.dat
Thread count = 1   m = 40000   n = 40000   overall time = 66.796288   computation time = 11.101893
```

A. Execution Time

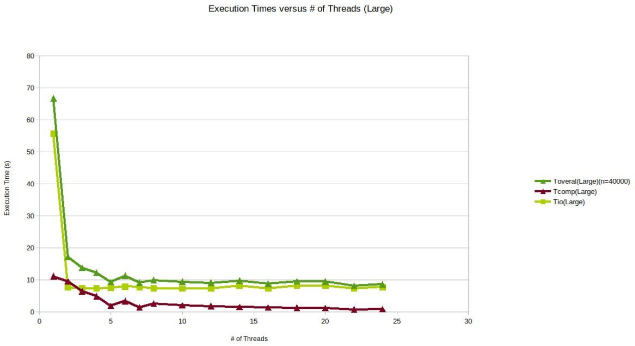


Fig.1 Observed execution times for 40000 x 40000 matrix

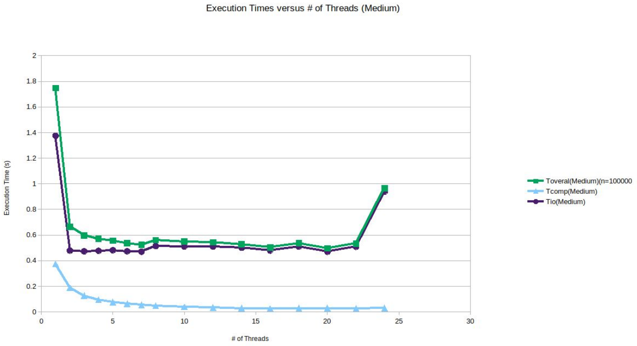


Fig.2 Observed execution times for 10000 x 10000 matrix

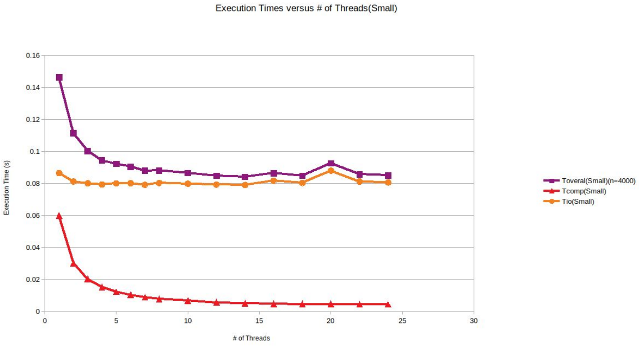


Fig.3 Observed execution times for 4000 x 4000 matrix

To collect variable data, we ran the code on small, medium, and large matrices. The small matrix was 4000 x 4000, the medium matrix was 10k x 10k, and the large matrix was 40k x 40k. For each matrix size, we ran the code on 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 18, 20, 22, and 24 threads. As can be seen in figures 1, 2, and 3, the overall time very quickly shrinks, leveling out at about 8 threads. This makes sense as the majority of the time is being spent on the multiplication, and the multiplication was parallelized, so that amount of time is shrinking quickly; however, there is also file I/O and thread spawning overhead that accounts for the time that is not reduced through parallelism. This overhead explains the outlier plots from the overall time on the large matrix multiplication and the I/O time on the large matrix multiplication given one thread.

B. Speedup

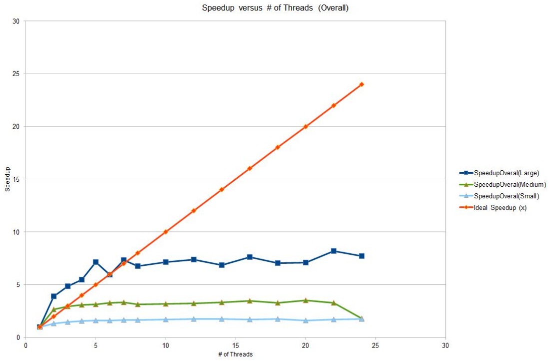


Fig. 4 Observed speedup for overall time for 3 distinct matrix sizes

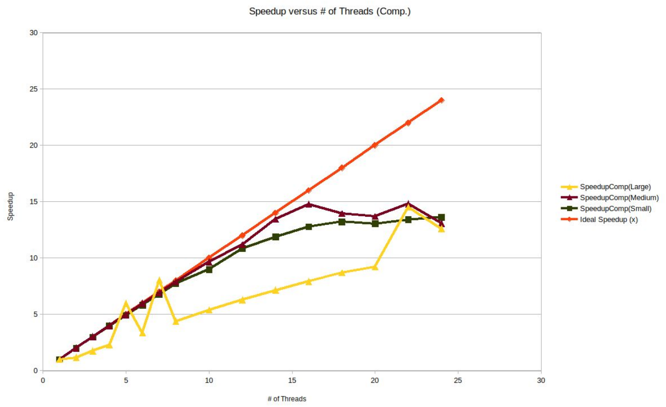


Fig. 5 Observed speedup for computation time for 3 distinct matrix sizes

For each matrix size we measured the speedup of the overall time and the computation time. As figure 4 and 5 shows, the greatest overall speedup was observed for the large sized matrix. This makes sense, because the large sized matrix causes the computation time to occupy a larger portion of the total time, allowing higher thread counts to reduce the total time by a greater amount.

C. *Efficiency*

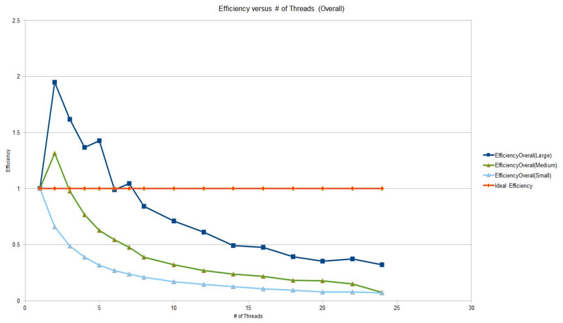


Fig. 6 Observed efficiency of overall time for 3 distinct matrix sizes

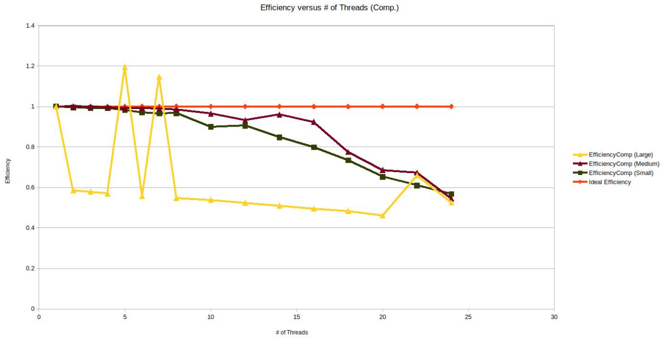


Fig. 7 Observed efficiency of computation time for 3 distinct matrix sizes

For each matrix size we calculated the efficiency of both the computation time and the overall time. As figure 6 and 7 shows, running the program with the large matrix was the most efficient overall. In fact, for small numbers of threads it actually exceeded ideal efficiency; this can happen for reasons beyond the scope of this paper, but more information can be found in the work cited here.[3]

IV. Conclusion

After compiling all the data from COMET and looking at the plots, it is clear that putting in the effort to parallelize the code for this problem benefits the performance greatly. Even with the overhead of the I/O, the performance from the computation was greatly improved. Even more so, using shared memory via multithreading allowed the parallel code to run very efficiently, particularly for large inputs. While writing parallel code is certainly difficult, or in some cases not possible, and running code in parallel causes otherwise unnecessary overhead, when a problem can benefit from parallelism the results speak for themselves. For problems like this where breaking up the work is fairly simple, it can cause massive speedups when running on just two or three threads.

References

[1] *R. Robey and Y. Zamora, Parallel and High Performance Computing, 1st ed., vol. 1. Shelter Island, NY: Manning Publications, 2020*

[2] *M. J. Quinn, Parallel programming in C with MPI and OpenMP. Boston, Mass.: McGraw-Hill, 2008.*

[3] *S. Ristov, R. Prodan, M. Gusev, and K. Skala, “Superlinear Speedup in HPC Systems:why and when?,” Annals of Computer Science and Information Systems, 2016. [Online]. Available: https://annals-csis.org/Volume_8/pliks/pliks/498.pdf. [Accessed: 24-Apr-2020].*