CS 4379 Parallel Programming   
Pthreads Programming Matrix Vector Calculation

Jacob Zahn

Computer Science, Texas Tech University

**This paper will compare computation times and speed ups for the guass.c program written for project 2. My first implementation of the project was to use OpenMP lazily to split the outer loop of gaussian elimination. While this solution is easy to implement it doesn’t provide the best performance attainable for this problem because it doesn’t balance the load of work that each processor must do. Reading the book [1], at the end of chapter 8 the author hints that a better algorithm to start with when attempting to parallelize this problem would be column wise LU factorization. This is the algorithm I attempted to parallelize in my implementation of guass.c.**

1. PTHREAD IMPLIMENTATION OF LU FACTORIZATION

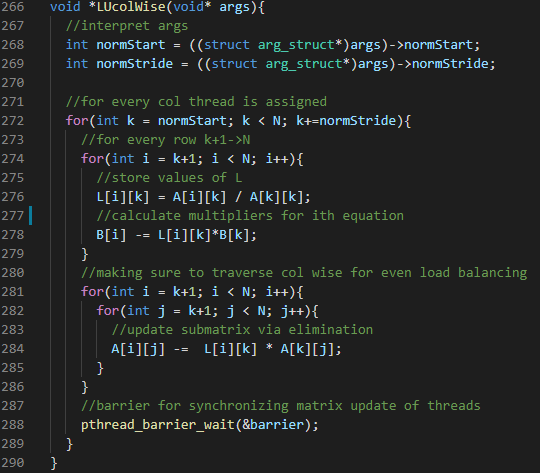


Fig. 1 Algorithm for column wise LU factorization

Fig. 1 shows the code used for the LU factorization in my implementation of guass.c. The outer most for loop uses a start point and a stride as a way of implementing cyclic column wise partitioning of LU factorization. This algorithm differs from gaussian elimination in that its performed column wise instead of row wise. I also use an additional global variable L[N][N] to store the Lower triangle matrix of A. This variable is not necessary as L can be stored in A as was done in the algorithm provided, the matrix is there for aiding my understanding of the algorithm. The Upper triangle of A is stored inside A.

A barrier is used to synchronize threads after they update the matrix A, so that there isn’t a chance a thread reads a section of the matrix that hasn’t been updated. The downside of using the barrier in this simple of an algorithm is that it can only run when the matrix dimensions are evenly dividable by the number of threads. Therefore, the following figures and their data feature N:1984, because it’s the largest evenly dividable by 32 number that is less than 2000 the hard-coded max matrix size.

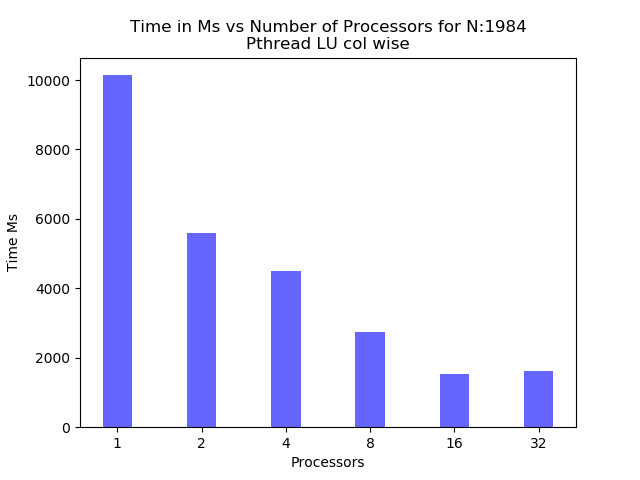


Fig. 2 Time in milliseconds for execution of program

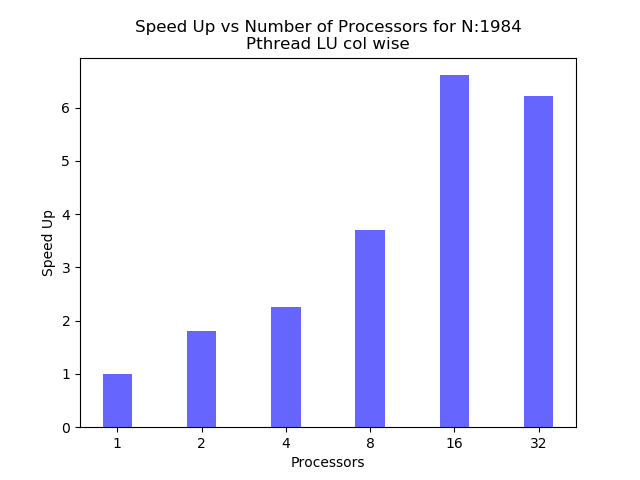


Fig. 3 Speed Up for execution of program

The time it took for each execution of the program decreased when using more threads until 32 as shown in Fig. 2. The speedup for increasing threads is sub linear as shown in Fig. 3. Overhead from managing the threads exceeds the speed up gained from additional threads when the program is ran using 32 threads. This may be because 1984 divided by 32 is only 62. 62 columns might not be enough work to warrant the number of threads.

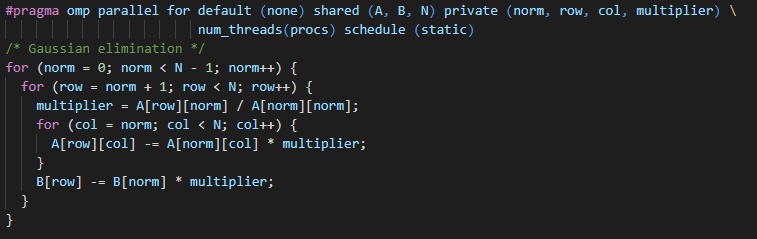


Fig. 4 shows a lazy OpenMP parallelization of gaussian elimination

Parallelizing gaussian elimination like shown in Fig. 4 will result in less than optimal speed up and performance. The Pthread equivalent program would implement row wise parallelization of the outer most loop of gaussian elimination, which does not have optimal load balancing of the work.

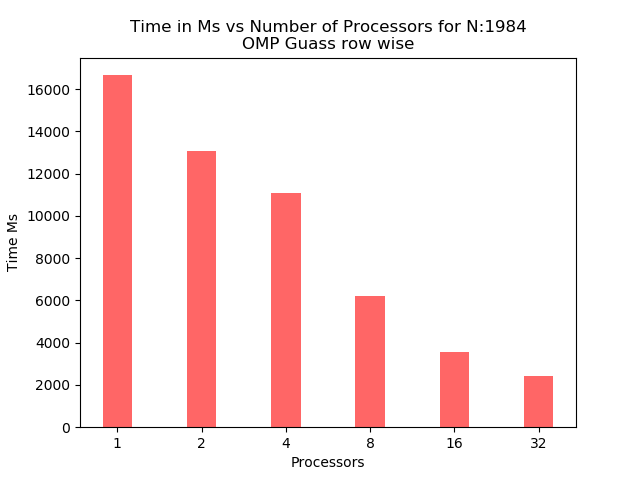


Fig. 5 Time in milliseconds for execution of program

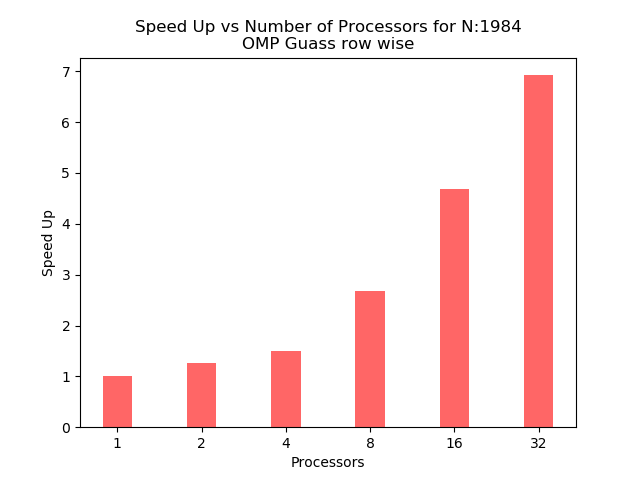


Fig. 6 Speed Up for execution of program

The serial execution of row wise gaussian elimination appears as shown in Fig. 5 to run longer than the serial execution of LUcolWise() as shown in Fig. 2. This may be because of overhead from OpenMP’s implementation of parallelization. Speed up is sub linear as shown in Fig. 6. However, the OpenMP program does make better use of the 32 threads as compared to my Pthread implementation shown in Fig. 3

Overall time of execution for my implementation as shown in Fig. 2 is lower compared to the overall time of execution for the OpenMP program as shown in Fig. 5 across all categories of threads.

REFERENCES

[1] Grama, A., 2003. Introduction to Parallel Computing, Second Edition. 2nd ed. Harlow: Addison-Wesley.