

Parallel Implementation for Gaussian Elimination Using MPI and OpenMP

Michigan State University, Department of Chemical Engineering and Material Science

Danqi Qu

Introduction

The purpose of this project is to develop an algorithm with parallel strategies for high efficiency Gaussian elimination. At the end, strong scaling and weak scaling has been tested to verify the degree of implementation.

Gaussian elimination is a canonical solver for linear systems of equations in linear algebra. Linear systems can be used to numerically solve partial differential equation, dense linear equations and so on. A sample of domains using dense linear equations are fusion reactor modeling, aircraft design and acoustic scattering. Any improvement in the time to reach the solution for linear system has a direct impact on the execution time of these applications. [1] There are three row operations involved in the algorithm, swapping two rows, multiplying a row with a nonzero number and adding a multiple of one row to another. The goal of doing these three operations is to get an upper triangle matrix. Then, either backward substitution or backward elimination (to get a diagonal matrix) can be done to reach the solution.

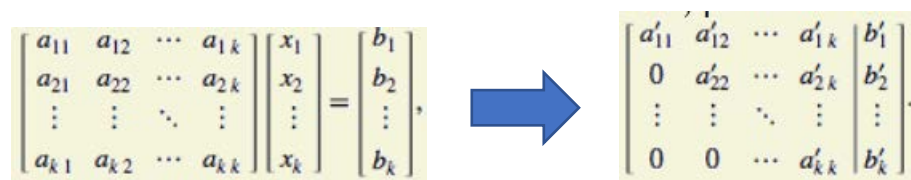
$$Ax = B$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix}, \quad \rightarrow \quad \begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1k} \\ 0 & a'_{22} & \cdots & a'_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{kk} \end{bmatrix} \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_k \end{bmatrix}.$$

Figure 1. Gaussian elimination procedure

There are mainly three different sequential algorithms, the KJI-SAXPY version, the JKI-GAXPY version and the IJK-DOT version, which are mentioned in this paper [2] for performing Gaussian elimination. In the last few years, the algorithm has received a lot of attention in attempt to improve its parallel performance. Based on these sequential algorithms, numerous studies of how to parallelize the computation and run on multicore machine has been published, [3] Such as the folding method of Evans and Hatzopoulos, bi-directional Gaussian elimination algorithm [4] and meet in the middle method [5]. In this project, the most straight forward way, KJI-SAXPY version, has been chosen to be modified into parallel version.

Methods

Mainly two strategies, domain decomposition and pipeline communication, have been applied to accomplish the parallel implementation.

It is very easy to think of distributing groups of rows or columns to different processors and each processor does its local computation simultaneously. The algorithm distributing rows is named row-wised parallel, correspondingly, distributing columns is named column-wised parallel. Here I chose row-wised parallel methods. The most intuitive way to share the work is to assign a group of contiguous rows to each processor. For instance, if there are p processors and the matrix size is n by n , processor 0 is responsible for the computation of row 1 to row n/p and processor 1 is assigned row $n/p+1$ to $n/p + n/p$ and so on. This method is called blocked mapping. But the problem is that the actual computation work is not balanced. As illustrated in figure 2, the left side represents the situation of blocked mapping. Due to the elimination process advancing, the amount of computation of processor assigned with low-numbered rows reduces more than that with high-numbered rows because the high-numbered rows need to do more elimination than low-numbered rows. The dark color represents the working time and the light color represents the waiting time.

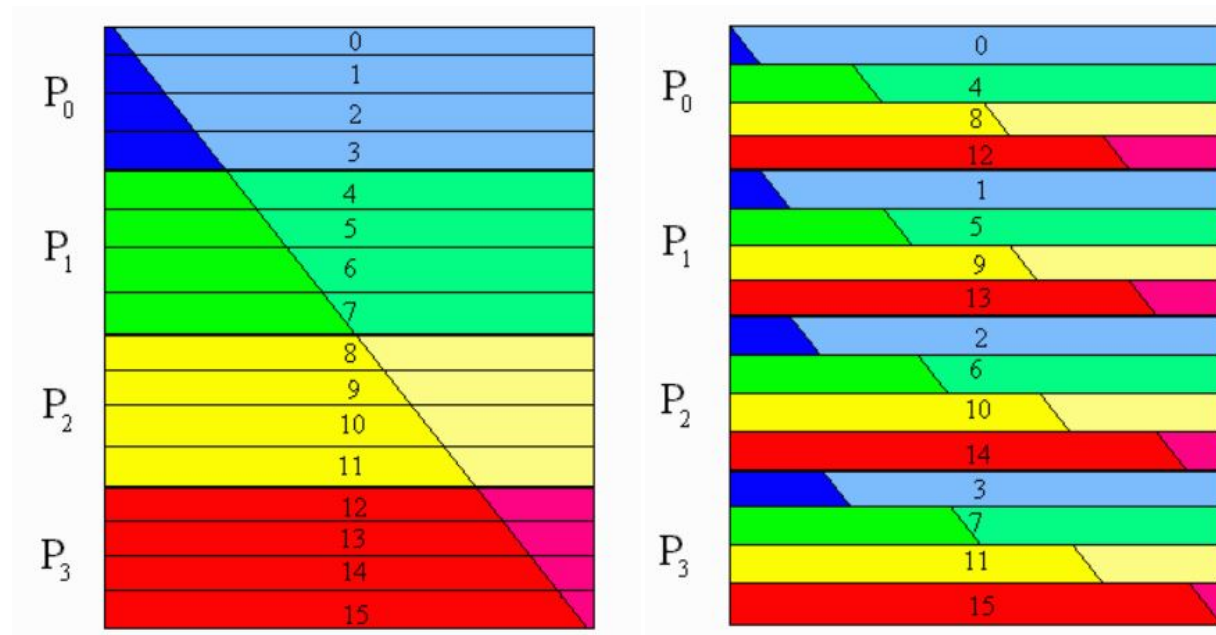


Figure 2. the load balance for blocked mapping and cyclic mapping [6]

The solution to the load balance is to do the cyclic mapping which means distribute the rows to processors periodically. For instance, assuming we have p processors in total, then the P_0 is assigned row 1, P_1 is assigned row 2, ..., P_{p-1} is assigned row p , P_0 is assigned row $p+1$ and so on. This can make the work divided more evenly than blocked mapping and can reduce the waiting time for each processor which significantly improve the load-balancing problem.

The other strategy of parallel implementation is to overlap the computation time with the communication time, which is called pipelined communication. As illustrated in figure 3, for broadcast communication, at every iteration, the processor which is holding the pivot row has to do a broadcast to send the pivot row to all other processors. The pivot row is the row needed at each specific elimination iteration. For instance, at iteration 1, the pivot row is the first row of matrix A and all other rows have to be added by the pivot row times a specific nonzero number to eliminate column 1. After the broadcast, the processors can do their local computation. The algorithm

contains such a sequence, broadcast elimination then repeat since each processor has to wait all the other processors finishing their local computation before doing a new broadcast. One possible way to implement the communication is doing the pipelined communication. Starting from the pivot processor, it sends the pivot row to its next processor and from then on each processor receives the pivot row from last processor and sends it to the next. The benefit of pipelined communication is that no rank has to wait for others. Once a rank receives the pivot row, it will send the row to the next rank and then begins its local computation. The processors no long need to wait for others finishing their own computation.

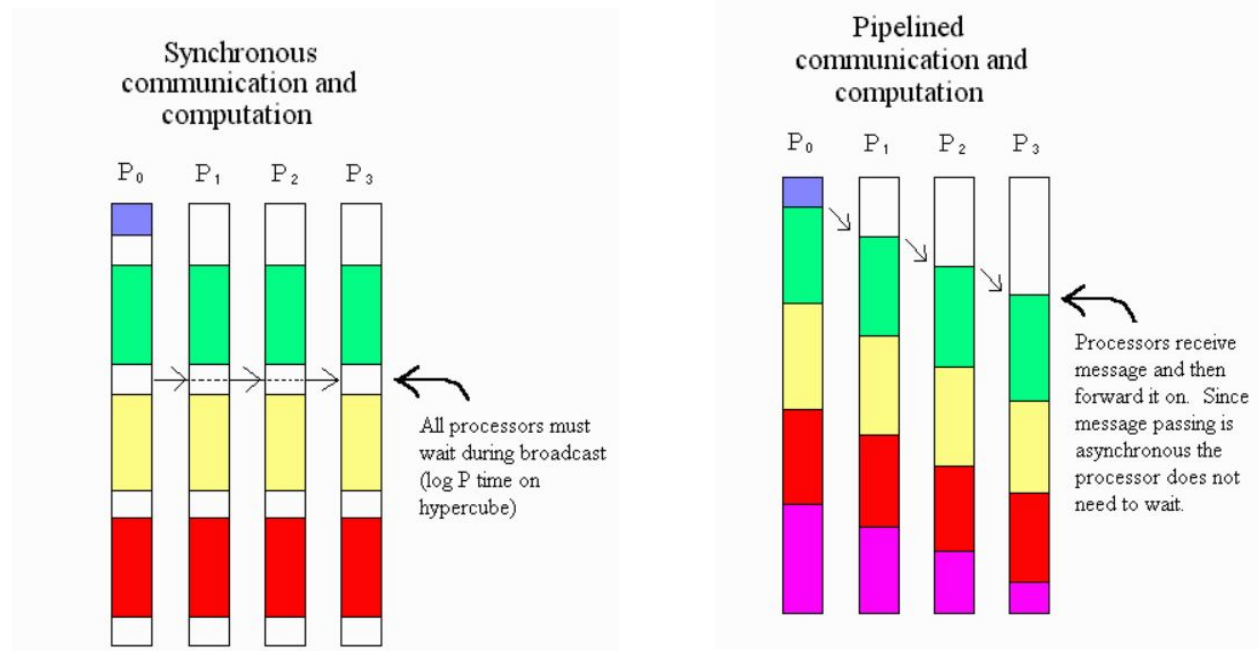


Figure 3. Broadcast communication and pipelined communication [6]

Unfortunately, the cyclic mapping complicated the pipelined communication since the starting processor is different at every iteration. The communication is accomplished by MPI_Send and MPI_Recv. In this project, non-blocking communication has not been tested yet.

As for OpenMP, a naïve implementation has been applied. Because each iteration of the inner loop i from $\text{pivot}+1$ to n is independent, and they don't have data dependency either, OpenMP loop parallelism is capable. The default chunk size and static schedule is used for this part. Loop i from $\text{pivot}+1$ to n is doing the elimination for row numbered $\text{pivot}+1$ to n . The innermost loop is operating the elements from $\text{pivot}+1$ to n of a specific row i .

```

do pivot = 1, (n-1)
!$omp parallel do private(xmult) schedule(runtime)
do i = (pivot+1), n
xmult = a(i,pivot) / a(pivot,pivot)
do j = (pivot+1), n
a(i,j) = a(i,j) - (xmult * a(pivot,j))
end do
b(i) = b(i) - (xmult * b(pivot))
end do
!$omp end parallel do
end do

```

Figure 4. OpenMP implementation for the inner loop. [7]

The meaning of every variable.

ierr: Fortran MPI ierror.

myRank: identify each rank, the number of process.

numRanks: give how many processes there are in all.

integer i, j, k: iteration counters.

nglobal: the size of matrix A, how many rows there are in all.

nlocal: record how many rows has been distributed to each rank.

mtrxA: store the elements of matrix A, mtrxA1 is for serial subroutine use.

mtrxB: store the elements of matrix B, mtrxB1 is for serial subroutine use.

mtrxX: store the elements of solution X, mtrxX1 is for serial subroutine use.

mtrx_A: store the local elements of partial matrix A for each rank.

mtrx_B: store the local elements of partial matrix B for each rank.

mtrx_X: store the local solution for each rank.

tempA: store the pivot row for each rank.

tempB: store the elements of matrix B corresponding to the pivot row for each rank.

time1: record the starting time of parallel algorithm.

time2: record the ending time of parallel algorithm.

timef: the difference between time1 and time2, report the total running time of the parallel algorithm.

diff: the difference between the solution X from the serial solver and the solution X from the parallel solver.

flna: the name of file stored matrix A.

flnb: the name of file stored matrix B.

fs: the number after the file name.

subroutine serial_gaussian: the subroutine for performing Gaussian elimination in serial.

subroutine InPut2D_F: the subroutine for inputting 2D array from a specific file.

subroutine InPut1D_F: the subroutine for inputting 1D array from a specific file.

subroutine OutPut1D_F: the subroutine for outputting 1D array to a specific file.

Results

The code has been tested on the HPC of MSU, development node intel18. The programming language is FORTRAN 90.

The matrix A and B are generated randomly by another code outside of the Gaussian elimination one. The input is the matrix A and B. The output is the solution matrix X. Both matrix A and matrix B are read in by external data and are first stored on process 0. It is then distributed to different processes row by row. Therefore, in addition to the rank 0, other processes only need to store the parts of the matrix A and B in their memory, where the storage space is $(n/p+1) * (nglobal+1)$ floating point numbers. The rank 0 needs to store the entire matrix A and B. Since the amount of data coming in and out is small, the "txt" files are used here as the input and output objects.

The weak scaling is tested by fixing the number of rows assigned to each rank. Here I set the parameter as 128 rows per rank which means the total size of matrix A is 2048 if 16 ranks have been involved. Both cyclic mapping and blocked mapping algorithm are tested. The results are presented in figure 5. As the number of processes increasing, the total run time increases linearly for both two mappings. The performance of cyclic mapping is slightly better than that of blocked mapping.

The strong scaling is tested by fixing the size of matrix A. Here matrix size 2048 is set for the whole testing. Before process number reaching 16, the total running time decreases linearly as the number of processes increasing. But after that, the total running time increases slightly. That maybe because the communication time increases as more processes involved and becomes dominate. The performance of cyclic mapping is also better than blocked mapping as shown in figure 6.

The figure 7 shows how the running time trends as the OpenMP threads increasing. The running time first reduces before OMP_NUM_THREADS equals to 4 and it slightly increases as the OMP_NUM_THREADS reaches 8. The reason for that abnormal increasing is still unknown.

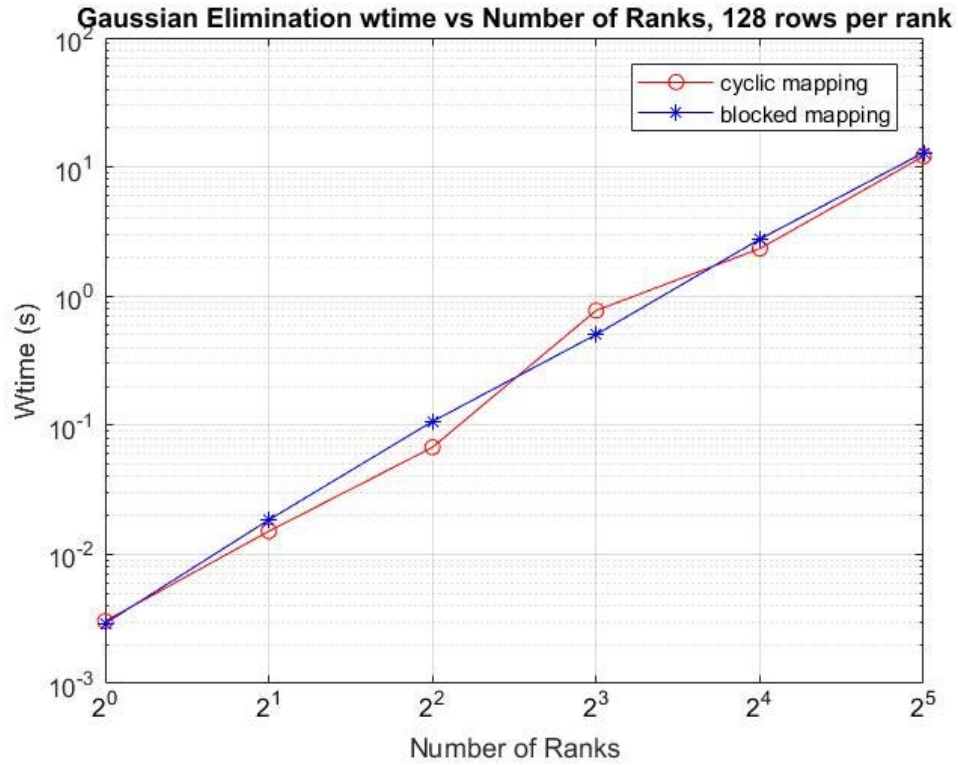


Figure 5. Weak scaling fixing the number of rows per rank

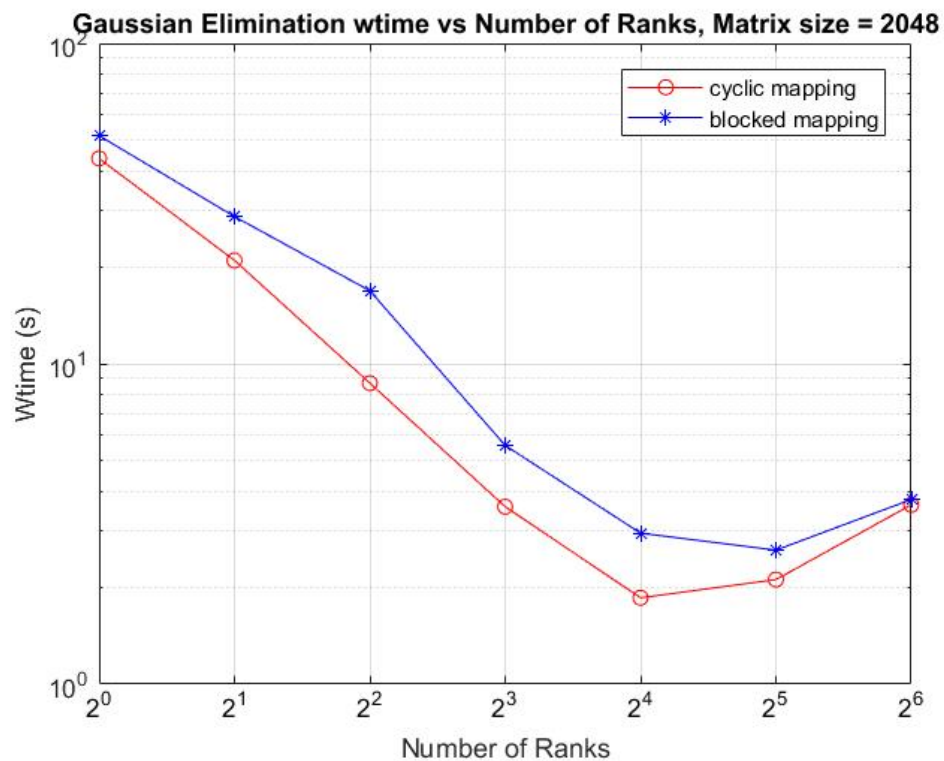


Figure 6. Strong Scaling fixing the size of matrix A

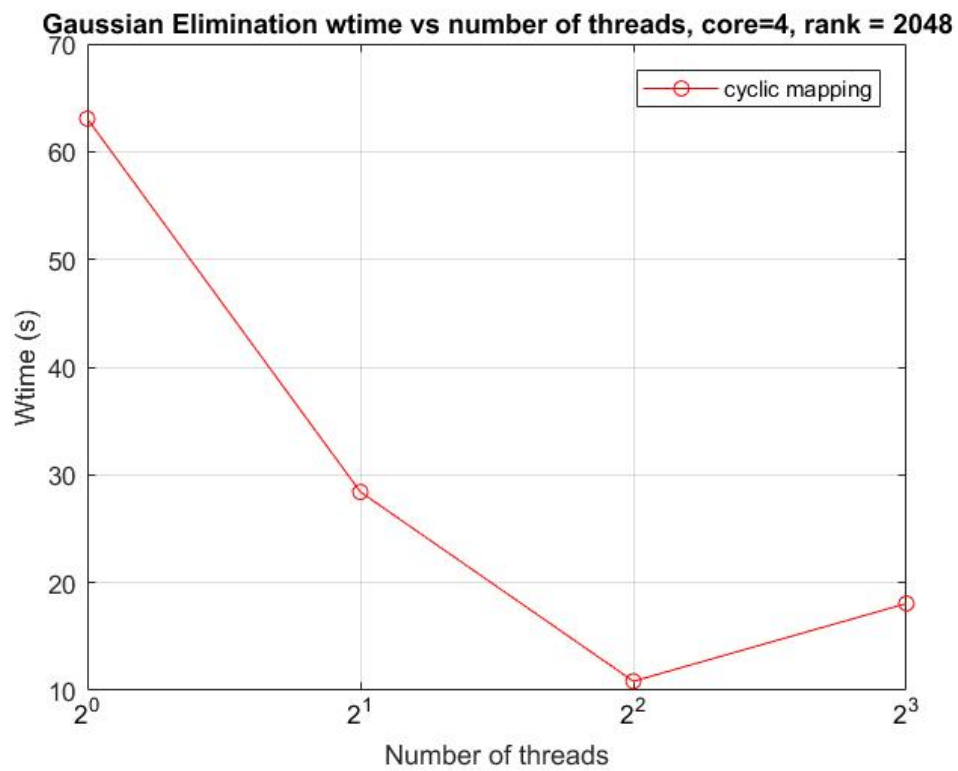


Figure 7. OpenMP threads speed-up

Conclusions

OpenMP and MPI are used to improve the serial gaussian elimination algorithm. From the results of weak scaling and strong scaling, MPI modified code has significant implement running on multicore machine. Domain decomposition plays an essential role in the implementation. The linearity of strong scaling proof that the algorithm has been well-paralleled. The cyclic mapping makes the load balanced in some degree. The pipeline communication does not perform as well as expected. OpenMP loop parallelism can also improve the performance of the parallel algorithm. In the future, non-blocking communication can be tested to see if it is beneficial to the performance. Comparison of broadcast communication, blocking communication and non-blocking communication should be made on the same graph. Additionally, other sequential algorithms such as GAXPY and DOT can also be modified into parallel versions.

References

- [1] M. M. Chawla, A parallel Gaussian elimination method for general linear systems, Intern. J. Computer Math., Vol. 42, pp. 71-82, 1991.
- [2] M. Cosnard, Parallel Gaussian elimination on an MIMD computer, Parallel Computing 6 (1988) 275-296.
- [3] Simplicio Donfack, A survey of recent developments in parallel implementations of Gaussian elimination, Concurrency Computat.: Pract. Exper. 2015.
- [4] S. F. McGinn and R. E. Shaw, Parallel Gaussian Elimination Using OpenMP and MPI, University of New Brunswick.
- [5] Fadi N. Sibai, Performance modeling and analysis of parallel Gaussian elimination on multi-core computers, Journal of King Saud University (2014) 26. 41-54.
- [6] <http://cseweb.ucsd.edu/classes/fa98/cse164b/Projects/PastProjects/LU/abstract.html>
- [7] https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html