COMP328 Assignment Report

Some Notations

N_cons	Number of constraints of the linear system
N_var	Number of variables of the linear system
Size of a linear system	N_cons * N_var
comm_size	The total number of processors used in MPI
rank(..)	The rank of a matrix ..
N_core	Number of threads of openMP or number processors of MPI

1. Gaussian-Jordan Elimination

1.1. Introduction

Gaussian-Jordan Elimination is a method that can be used to solve a system of linear equations or find the inverse of an invertible matrix. It allows three kinds of operations, namely i) swap two rows, ii) multiply a row by a nonzero scalar, and iii) add a scalar multiple of one row to another row[1]. The method will transform an input two-dimensional matrix into its reduced row echelon form, which means the output matrix should satisfy that I. All rows consisting of only zeroes being at the bottom, II. The leading nonzero coefficient (pivot) of a nonzero row is always strictly to the right of the pivot of the row above it, III. All pivots are one, and IV. Each column containing a pivot has all its other entries being zero [2].

This program uses systems of linear equations $Ax = b$ as input, combines the coefficient matrix A and constant variables b into an augmented matrix $M = A|b$, conducts Gaussian-Jordan Elimination on M, and then outputs the result of the linear system. The value of $rank(M)$, $rank(A)$, N_var are used to determine whether the linear system is infeasible, or feasible with a unique solution, or feasible with infinitely many solutions [3] (explained in *Appendix A*). If the linear system has a unique solution, the program will print out the first N_var entries in the last column of the output M as the values of the variables.

This program provides three versions of the Gaussian-Jordan Elimination method: serial (gaussian_serial.c), parallelized with openMP (gaussian_OMP.c), and parallelized with MPI (gaussian_MPI.c). (Other assistant functions are in gaussianLib.c.) The two parallelized versions share the same main ideas as the serial version in terms of the elimination algorithm.

1.2. Serial Algorithm

1.2.1. Main Idea

Input: 2-D Matrix *M*

a. **Find Pivot**: find the left-most non-zero element (to the right of the previous pivot if it's not the first round) as the new pivot
b. **Swap**: swap the pivot row to the row just under the previous pivot row (to the top row if it's the first round)
c. **Reduce to One**: reduce the pivot to one by dividing pivot row by pivot
d. **Clear Columns**: clear all the elements in pivot column to zero except pivot itself by reducing each row by some multiple of pivot row
e. **Repeat**: Start a new round by repeating the above four steps until no new pivot can be found

Ouput: *M* in reduced row echelon form

1.2.2. Pseudo Code

```
void Gaussian_serial(double M[N_row][N_col], int* rank_A, int* rank_M){
    int row_flag = 0, col = 0, row = 0, j = 0, i = 0;
    for(col = 0; col < N_col; ++col){
        for(row = row_flag; row < N_row; ++row){
            // a. Find Pivot
            if(M[row][col] != 0.0) {
                // b. Swap
                swap_two_rows(M, row, row_flag);
                // c. Reduce to One
                for(j=N_col-1; j>=col; --j){    M[row_flag][j] = M[row_flag][j] / M[row_flag][col];    }
                // d. Clear Columns
                for(i=0; i<N_row; ++i){
```

```
                              if(i!=row_flag &&    M[i][col]!=0){
                                    for(j=N_col-1; j>=col; --j){    M[i][j] = M[i][j] -    M[i][col] * M[row_flag][j];    }
                        }     }
                        ++ row_flag;    *rank_M=row_flag;    *rank_A= col!=N_col-1 ? row_flag:row_flag-1;
                        break;
}     }     }     }
```

Table 1: Gaussian_serial

1.2.3. Analysis & Limitations

The above algorithm has 4 levels of for-loops. The first level (outer) for-loop is to search for the pivot from left to right. The elements in the matrix will change when the loop goes on. The rows may be swapped. All rows may change based on the pivot column. The next round of the loop is based on the whole resulted matrix from the previous round. Thus, the outer for-loop has data dependence in terms of its round.

The second level for-loop is to search for the pivot from top to down for each column. The code inside will only be executed at most once for each column since a single column can only have at most one pivot.

The third level has two for-loops. One is to reduce the pivot to one. It only cares about the data within the pivot loop. The other has the fourth level for-loop nested inside. It is to clear other elements in the pivot column to zero. It concerns all rows basically.

Limitations of using the serial algorithm are high time complexity and high space complexity. With the 4 levels of for-loops, the average time complexity reaches $O(N\_cons \times N\_var^2)$. The space complexity reaches $N\_con \times N\_var$. This makes it difficult to deal with matrices of a large size.

2. Implementation

2.1. OpenMP Version

As stated above in Section 1.2.3., the first two levels of for-loops cannot be parallelized by openMP, because the first level for loop has data dependence on the previous rounds, and the computation inside the second level for-loop will still be executed by only one single thread.

The two for-loops at level three can be parallelized since the data is independent in terms of the column or row computation. However, the first for-loop at level three only handles one row. The computation is not complex and after testing it is found that the overhead of parallelizing this part is large and may lead to an increasing amount of run time.

The second for-loop is suitable for parallelizing since the computation concerns all rows. After testing it is found that parallelizing at the fourth level for-loop within this for-loop causes an increasing amount of run time, which may be due to the high cost of the communication time. It is also found that parallelizing the second for-loop at the third level as a whole leads to a decreasing amount of time.

Therefore, " *#pragma omp parallel for default(none) private(i, j) shared(M, N_col, col, row_flag, N_row)* " is put above the second for-loop at the third level, which has been marked in blue in *Table 1*. The threads have private variables *i* and *j* as the index of the for-loops, and share all other variables. The matrix *M* contains the elements to update and these elements have no data dependence, so *M* can be shared. The variables *N_col, col, row_flag, N_row* will not be updated in the loops, so they can also be shared. The threads are in work-sharing mode and each handles the computation of one or multiple rows.

Except for the *#pragma* mentioned above and the code for recording the run time, the code in the openMP version remains the same as the serial version.

2.2. MPI Version

The MPI Version algorithm still shares the same main ideas as the serial version. The part of computation for parallelizing is still at step d. Clear Columns similar to the openMP version does. However, the whole input matrix *M* is scattered evenly to each processor, and the segments of *M* is stored in *M_seg* at each processor. *M_seg* contains *N_cons/comm_size* rows of M. Thus, this algorithm asks the number of rows of *M* to be divisible by the number of processors.

Step a. Find Pivot is done by the processors (*row_processor*) that contain the target rows until a new pivot is found. Then the flag *find_pivot* will be set to one and broadcast to all the processors. Step b. Swap needs row data exchange between the row_processor and the processor (*target_processor*) containing the row where the pivot row should be at after being swapped. Step c. Reduced to One is done by *target_processor* only. Then it will broadcast the updated pivot row (stored in *pivot_line*) to all processors, so that all processors can update the rows to finish step d. Clear Column based on the *pivot_line*.

Communications are needed at the beginning when the root processor scatters the input *M*, at the end when the root processor gathers M_seg to form the output, and at each time broadcasting *find_pivot* or swapping rows from different processors or broadcasting the *pivot_line* is needed. Table 2 below shows the pseudo code of this algorithm.

```
void Gaussian_MPI(double M[N_row][N_col], int* rank_A, int* rank_M){
        int part_size = N_cons * (N_var+1) /comm_size, row_range = N_cons / comm_size;
        int start_row = my_rank * row_range, end_row = (my_rank+1) * row_range - 1
        double M_seg[N_cons][N_var+1], pivot_line[N_var+1], temp_line[N_var+1];
        int row_flag = 0, local_flag = 0, local_row = 0;
        int find_pivot = 0, target_processor = 0, row_processor = 0;
        int col, row, i,j;
        // scatter M
        MPI_Scatter(M, part_size, MPI_DOUBLE, M_seg, part_size, MPI_DOUBLE, root, MPI_COMM_WORLD);
        for(col = 0; col < N_col; ++col){
                for(row = row_flag; row < N_row; ++row){
                        // the processor having the "row"th row
                        row_processor = (int) floor(((double)row / (double)row_range));
                        find_pivot = 0;
                        // a. Find Pivot
                        if(my_rank == row_processor){
                                local_row = row % row_range;
                                if(M_seg[local_row][col] != 0){    find_pivot = 1;    }
                        }
                        // broadcast find_pivot
                        MPI_Bcast(&find_pivot, 1, MPI_INT, row_processor, MPI_COMM_WORLD);
                        if(find_pivot == 1){
                                // the processor that the pivot row should be in (after swapping)
                                target_processor = (int) floor(((double)row_flag / (double)row_range));
                                local_flag = row_flag % row_range;
                                // b. Swap
                                if(row != row_flag){
                                        if(target_processor == row_processor){
                                                if(my_rank == target_processor){    swap_two_rows(M_seg, local_flag, local_row);    }
                                        }else{
                                                if(my_rank == target_processor){
                                                     MPI_Sendrecv(M_seg[local_flag], N_col, MPI_DOUBLE, row_processor, 0,
                                                                  M_seg[local_flag], N_col, MPI_DOUBLE, row_processor, 1,
                                                                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                                                }
                                                if(my_rank == row_processor){
                                                    MPI_Sendrecv(M_seg[local_row], N_col, MPI_DOUBLE, target_processor, 1,
                                                                  M_seg[local_row], N_col, MPI_DOUBLE, target_processor, 0,
                                                                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                                }       }       }
                                // c. Reduce to One
                                if(my_rank == target_processor){
                                        for(j=N_col-1; j>=col; --j){
                                                M_seg[local_flag][j] = M_seg[local_flag][j] / M_seg[local_flag][col];
                                        }
                                        memcpy(pivot_line, M_seg[local_flag], N_col * sizeof(double));
                                }
                                // broadcast pivot line
                                MPI_Bcast(pivot_line, N_col, MPI_DOUBLE, target_processor, MPI_COMM_WORLD);
                                // d. Clear Columns
                                for(i=0; i<row_range; ++i){
                                        if((my_rank * row_range + i != row_flag) && (M_seg[i][col] != 0)){
                                                for(j=N_col-1; j>=col; --j){    M_seg[i][j] = M_seg[i][j] - M_seg[i][col] * pivot_line[j];    }
                                }       }
                                ++ row_flag;    *rank_M=row_flag;    *rank_A= col!=N_col-1 ? row_flag:row_flag-1;
                                break;
        }       }       }
        // gather M_seg
        MPI_Gather(M_seg, part_size, MPI_DOUBLE, M, part_size, MPI_DOUBLE, root, MPI_COMM_WORLD);
}
```

Table 2: Gaussian_MPI

3. Analysis

All the linear systems are generated randomly with the same random seeds. Therefore, the test cases of one *size (same N_cons and same N_var) have exactly the same input data. The maximum number of cores that can be requested via -c or -n is 40 in Barkla, so the largest number of threads or processors tested here is 40. The run time is measured five times in each situation and the median value is used here for analysis. A full description of all the submitted files can be found in Appendix B.*

3.1. Timing Region & Function

To see the influence of parallelizing, we need to record the time for the three versions of algorithms to complete the elimination, i.e., starting from processing the input matrix, and ending at outputting the matrix in reduced row echelon form. Hence, in terms of serial version and openMP version, the program will print out the time of completing the outer for-loop. In terms of the MPI version, the root processor will print out the time from beginning to scatter *M* to finishing gathering all *M_seg.* The start and the end of the timing region is highlighted in yellow in *Table 1* and *Table 2*.

Wall time is used to measure the run time of the program. Serial version uses *clock_gettime(CLOCK_REALTIME, &time)*. OpenMP version uses *omp_get_wtime().* MPI version uses *MPI_Wtime().* The time is measured in milliseconds.

3.2. Optimization Level

To find out a most-fit optimization level, a linear system of N_cons=240, N_var=240 is used to compare the run time of the algorithms on 1, 6, 20, 40 threads or processors based on the optimization level of -O0, -O1, -O2 and -O3. The related slurm output files are renamed with “opt-” at the beginning. The run time (milliseconds) in different situations are summarized as below:

	Serial	**OMP**				**MPI**			
N_core / **OPT**	1	1	6	20	40	1	6	20	40
0	28.01	46.30	21.60	12.72	41.73	65.49	12.21	8.25	6.28
1	4.35	15.11	8.71	27.03	59.37	10.81	4.24	5.53	5.81
2	6.13	11.08	7.39	8.04	35.87	5.86	3.72	5.17	5.87
3	2.22	10.92	7.50	8.85	30.59	5.47	3.49	4.89	5.82

Table 3: Optimization

It can be seen that the optimization has stronger influence when the number of cores is small. It is also shown that in most cases the run time decreases significantly when the optimization level changes from 0 to 1. The run times can still keep decreasing when the optimization level changes from 1 to 2, and usually stay at the same level or even increase a bit when the optimization level changes from 2 to 3. Therefore, -O2 is used in all the later tests.

3.3. Performance Analysis

3.3.1. Accuracy

There are three manually input testcases provided in *gaussianLib.c* for checking the correctness of the algorithms. The screenshots of their running results are put in *Appendix C*. Testcase 1 has unique solutions. Testcase 2 has infinitely many solutions. Testcase 3 is infeasible. All the three versions of algorithms obtained correct solutions of the three testcases, which means they should be able to deal with any input linear system correctly. The output precision is set to ".2f", i.e. two numbers after the decimal point. The precision can be changed in *gaussianLib.c* if needed.

Other testcases used below are larger than 50 * 50, which are too large to be printed out. The three versions of code obtain the same solution of these testcases. If the user needs to see the whole results, first open *gaussianLib.h* to change the defined value of *N_var* and *N_cons* as needed, then open corresponding *run_*.sh*, change the value of *size_row* and *size_col* equal to *N_var* and *N_cons* above, uncomment the line " *echo '============ print test case ================='* ", in the next line change the number after *./{EXE_*}* from *-2* to *2,* and finally run the bash file in command line by " *sbatch --exclusive -t 3 run_*.sh gaussian_*.c gaussianLib.c* ". (Don’t forget to change the "*" to the version name you need. The "-c" or "-n" has been pre-defined in the bash file.) Then the user can see the coefficient matrix *A*, the variables *b*, the constant *b*, the augmented matrix *M*, *M* in reduced echelon form, and the solution of the linear system in the slurm output file.

3.3.2. Space Efficiency

Both optimization level and parallelizing can affect space efficiency. Through testing, it is found that when “-

O0" is used all of the three versions of algorithm cannot handle linear system of size 600 * 600, which will lead to "Segmentation fault (core dumped)" probably due to the limitation of RAM. After changing to "-O3", the algorithms can even handle linear systems of size 720 * 720.

openMP is based on shared memory so it does not have much influence on the space needed. MPI is based on distributed memory. Although at last the whole *M* still needs to be gathered to one processor, which means the maximum space needed on one processor sill cannot be reduced, the space needed at only computation steps can be reduced by parallelizing from *sizeof(M) = N_var * N_cons * sizeof(double)* to *sizeof(M_seg) = sizeof(M) / comm_size.*

3.3.3. Time Efficiency

To compare the running time of the three versions of Gaussian Elimination, linear systems of size (N_cons * N_var) 60 * 60, 240 * 240, 480 * 480, 600 * 600, 240 * 600 and 600 * 240 are used to test the run time on 1, 3, 6, 10, 15, 20, 24, 30 and 40 threads or processors. *Table 4* below shows a summary of the test result. The time is measured in milliseconds. The optimal time in terms of the whole line, the optimal time in terms of one version of code, and the worst time in terms of the whole line are highlighted in light orange, blue and grey respectively. The curve knees (where the curve stops decrease sharply) of each line are marked in bold.

		Serial	openMP									MPI								
	N_core / Size	1	1	3	6	10	15	20	24	30	40	1	3	6	10	15	20	24	30	40
Square	60 * 60	0.06	4.07	4.51	4.46	4.51	6.20	4.42	8.73	10.08	18.71	0.34	1.94	2.04	2.33	1.87	3.02		2.59	
	240 * 240	5.66	10.87	8.54	**6.81**	6.32	6.67	10.49	23.00	20.90	26.91	5.85	**3.76**	3.78	4.52	3.99	5.44	4.73	4.98	5.72
	480 * 480	26.97	55.78	24.73	19.18	**14.95**	42.20	27.79	42.17	38.73	57.41	37.35	14.30	**10.91**	9.83	9.43	11.38	11.15	11.10	11.62
	600 * 600	44.21	98.08	44.53	30.89	**22.96**	20.38	18.27	17.18	47.10	72.40	66.85	23.75	17.19	**14.84**	12.59	14.72	13.25	13.02	13.24
Wide	240 * 600	17.72	28.15	15.4	**12.83**	12.03	11.44	22.34	26.67	22.26	37.98	18.5	**7.77**	6.39	6.35	6.18	7.14	7.49	6.61	7.49
Long	600 * 240	10.28	28.22	16.8	**12.82**	11.17	10.7	25.18	34.64	28.11	35.72	18.23	**7.76**	6.32	6.23	5.95	7.33	7.33	7.37	7.32

Table 4: Time Comparison

It can be clearly seen from the above table that the serial version has an absolute advantage over small linear systems (60 * 60). Its run time is almost 70 times quicker than the optimal time reached by openMP version and 5 times quicker than the optimal time reached by the MPI version. Parallelizing on small linear systems can not render a better run time since it can be clearly seen that the run time increases as the number of cores increases.

For linear systems of a middle size (240 * 240), parallelizing can decrease the run time a bit. The serial version needs 5.66 milliseconds. The optimal time of openMP version is 6.32 milliseconds using 10 threads. The optimal time of MPI is 3.76 milliseconds using 3 threads. The speedups of openMP and MPI compared to their version of code using N_core=1 are 10.87/6.32=1.72 and 5.85/3.76=1.55. The efficiency is 1.72/10=0.172 and 1.55/3=0.52 respectively.

For large linear systems (480 * 480 and 600 * 600), parallelizing can decrease the run time significantly. Taking the size 600 * 600 as an example, the serial version needs 44.21 milliseconds, the optimal time of openMP version reaches 17.18 milliseconds using 24 threads and the optimal time of MPI version reaches 12.59 milliseconds using 15 cores. The speedups of openMP and MPI comparing to their version of code using N_core=1 are 98.08/17.18=5.71 and 66.85/12.59=5.30. The efficiency is 5.71/24=0.23 and 5.3/15=0.35.

The run times on linear systems of the same amount of elements but with different shapes (long: 600 * 240; Wide: 240 * 600) are quite close. It seems that the running time depends on N_var and N_cons evenly without too much bias.

It shows clearly from *Table 4* that the run time usually drops sharply from 1 core to 6 cores, and tends to decrease slowly to reach the optimal, and then for openMP the run time usually increases quickly as the thread number continues to increase, or for MPI the run time usually remains at the same level or increase a little as the processor number continues to increase. Thus it seems that MPI is more stable than openMP when the number of cores goes beyond the optimal case.

It is also clearly displayed in *Table 4* that except for small linear systems, the MPI version always obtains the shortest run time, which is at 15 processors in most cases. However, the run time at the curve knee is very close to the optimal time. Thus, the efficiency at the curve knee can increase a lot with speedup only decreasing a bit if the threads number is changed from the optimal. Therefore, the MPI version with around 10 processors can be the best solution for Gaussian Elimination.

4. Conclusion

In summary, the program provides three versions of code: Serial version, OpenMP version and MPI version. Their implementation has been explained in section 1.2.1, section 1.2.2, section 2.1 and section 2.2. It is suggested to use -O2 when running the code. For linear systems smaller than 100 * 100, it is suggested to use the serial version. For other larger linear systems, it is suggested to use MPI version, because parallelizing can allow shorter run time for large matrix and MPI version usually performs better than openMP version in both stability and run time.

References

[1] T. P. S. University. (2022). *M.7 Gauss-Jordan Elimination* [Online]. Available: https://online.stat.psu.edu/statprogram/reviews/matrix-algebra/gauss-jordan-elimination.

[2] wikipedia, "Gaussian elimination," 2022.

[3] R. Blair. Class Lecture, Topic: “Math 240: Linear Systems and Rank of a Matrix”, University of Pennsylvania, Jan., 2011.

Appendices

Appendix A

N_var is defined according to each test case. The value $rank(M)$ and $rank(A)$ are computed when conducting Gaussian Elimination.

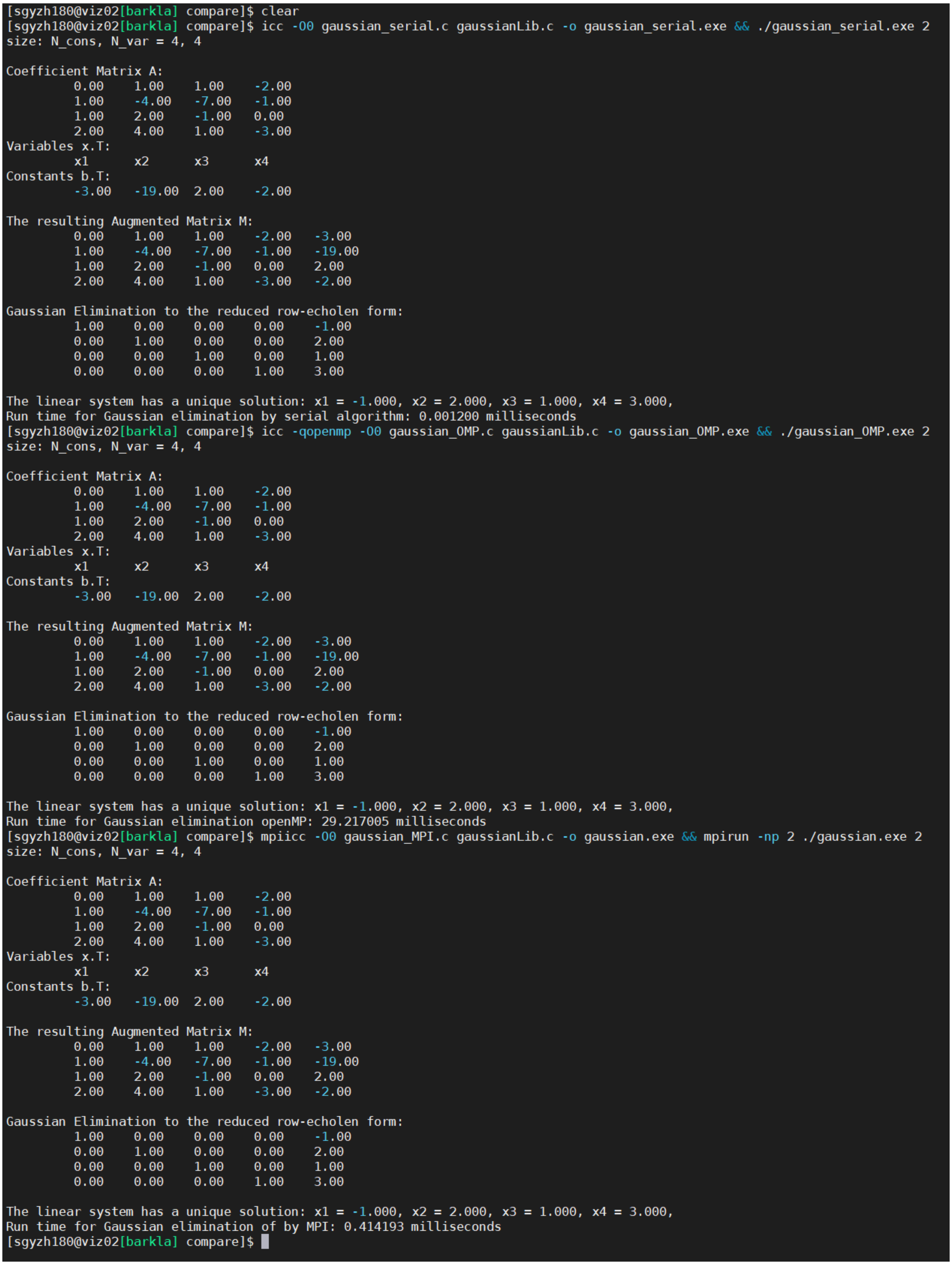
- The linear system has a unique solution. $\Leftrightarrow rank(A) = rank(M) = N\_var$
- The linear system has infinitely many solutions. $\Leftrightarrow rank(A) = rank(M) < N\_var$
- The linear system is infeasible. $\Leftrightarrow rank(A) < rank(M)$

Appendix B

File	Description
gaussian_serial.c gaussian_OMP.c gaussian_MPI.c	Main function and Serial / openMP / MPI version of gaussian elimination
gaussianLib.c gaussianLib.h	Other assistant functions and their headers
opt_serial.sh opt_omp.sh opt_MPI.sh	Bash files for comparing the run time of the optimization level of a given number of threads / processors and a given size of linear systems. The run time of each level will be recorded 5 times.
run_serial.sh run_omp.sh run_MPI.sh	Bash file for comparing the run time of a different number of threads/processors on a linear system of a given size. “-c” or “-n” defines the max number of cores. The file will record the time on 1~max number of cores each for 5 times.
opt-N_cons-N_var-[version]-N-slurm-...	The output file of *opt_[version].sh* on the linear system of size *N_cons * N_var* using *N* threads/processors
run-N_cons-N_var-[version]-N-slurm-...	The output file of *run_[version].sh* on the linear system of size *N_cons * N_var* using *N* threads/processors

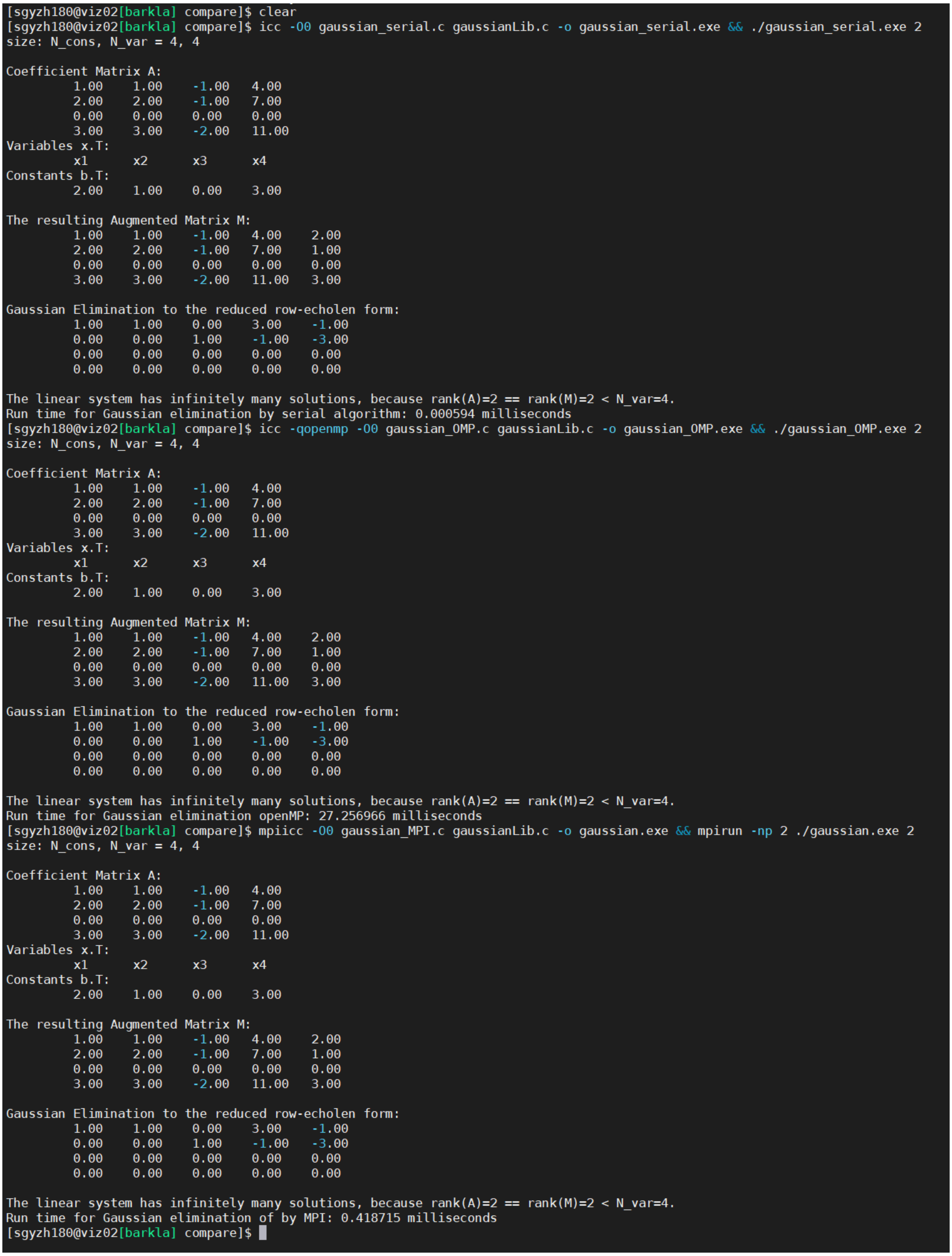
Appendix C

1. Test case 1 with a unique solution by Serial – OMP with 2 threads – MPI with 2 threads



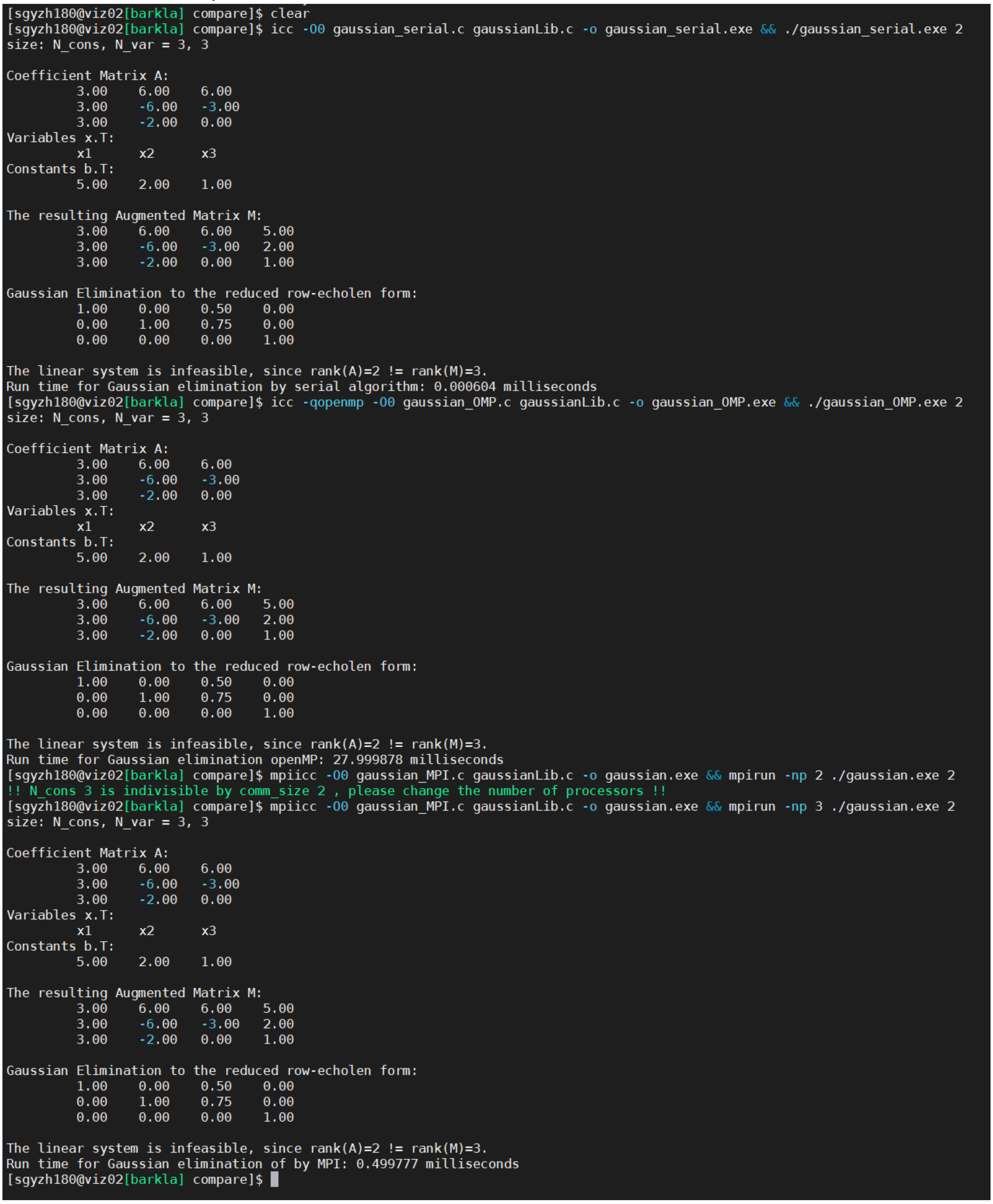
```
[sgyzh180@viz02[barkla] compare]$ clear
[sgyzh180@viz02[barkla] compare]$ icc -O0 gaussian_serial.c gaussianLib.c -o gaussian_serial.exe && ./gaussian_serial.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         0.00    1.00    1.00    -2.00
         1.00    -4.00   -7.00   -1.00
         1.00    2.00    -1.00   0.00
         2.00    4.00    1.00    -3.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         -3.00   -19.00  2.00    -2.00

The resulting Augmented Matrix M:
         0.00    1.00    1.00    -2.00   -3.00
         1.00    -4.00   -7.00   -1.00   -19.00
         1.00    2.00    -1.00   0.00    2.00
         2.00    4.00    1.00    -3.00   -2.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.00    0.00    -1.00
         0.00    1.00    0.00    0.00    2.00
         0.00    0.00    1.00    0.00    1.00
         0.00    0.00    0.00    1.00    3.00

The linear system has a unique solution: x1 = -1.000, x2 = 2.000, x3 = 1.000, x4 = 3.000,
Run time for Gaussian elimination by serial algorithm: 0.001200 milliseconds
[sgyzh180@viz02[barkla] compare]$ icc -qopenmp -O0 gaussian_OMP.c gaussianLib.c -o gaussian_OMP.exe && ./gaussian_OMP.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         0.00    1.00    1.00    -2.00
         1.00    -4.00   -7.00   -1.00
         1.00    2.00    -1.00   0.00
         2.00    4.00    1.00    -3.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         -3.00   -19.00  2.00    -2.00

The resulting Augmented Matrix M:
         0.00    1.00    1.00    -2.00   -3.00
         1.00    -4.00   -7.00   -1.00   -19.00
         1.00    2.00    -1.00   0.00    2.00
         2.00    4.00    1.00    -3.00   -2.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.00    0.00    -1.00
         0.00    1.00    0.00    0.00    2.00
         0.00    0.00    1.00    0.00    1.00
         0.00    0.00    0.00    1.00    3.00

The linear system has a unique solution: x1 = -1.000, x2 = 2.000, x3 = 1.000, x4 = 3.000,
Run time for Gaussian elimination openMP: 29.217005 milliseconds
[sgyzh180@viz02[barkla] compare]$ mpiicc -O0 gaussian_MPI.c gaussianLib.c -o gaussian.exe && mpirun -np 2 ./gaussian.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         0.00    1.00    1.00    -2.00
         1.00    -4.00   -7.00   -1.00
         1.00    2.00    -1.00   0.00
         2.00    4.00    1.00    -3.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         -3.00   -19.00  2.00    -2.00

The resulting Augmented Matrix M:
         0.00    1.00    1.00    -2.00   -3.00
         1.00    -4.00   -7.00   -1.00   -19.00
         1.00    2.00    -1.00   0.00    2.00
         2.00    4.00    1.00    -3.00   -2.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.00    0.00    -1.00
         0.00    1.00    0.00    0.00    2.00
         0.00    0.00    1.00    0.00    1.00
         0.00    0.00    0.00    1.00    3.00

The linear system has a unique solution: x1 = -1.000, x2 = 2.000, x3 = 1.000, x4 = 3.000,
Run time for Gaussian elimination of by MPI: 0.414193 milliseconds
[sgyzh180@viz02[barkla] compare]$ 
```

2. Test case 2 with infinitely many solutions by Serial – OMP with 2 threads – MPI with 2 threads



```
[sgyzh180@viz02[barkla] compare]$ clear
[sgyzh180@viz02[barkla] compare]$ icc -O0 gaussian_serial.c gaussianLib.c -o gaussian_serial.exe && ./gaussian_serial.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         1.00    1.00    -1.00   4.00
         2.00    2.00    -1.00   7.00
         0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         2.00    1.00    0.00    3.00

The resulting Augmented Matrix M:
         1.00    1.00    -1.00   4.00    2.00
         2.00    2.00    -1.00   7.00    1.00
         0.00    0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00   3.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    1.00    0.00    3.00    -1.00
         0.00    0.00    1.00    -1.00   -3.00
         0.00    0.00    0.00    0.00    0.00
         0.00    0.00    0.00    0.00    0.00

The linear system has infinitely many solutions, because rank(A)=2 == rank(M)=2 < N_var=4.
Run time for Gaussian elimination by serial algorithm: 0.000594 milliseconds
[sgyzh180@viz02[barkla] compare]$ icc -qopenmp -O0 gaussian_OMP.c gaussianLib.c -o gaussian_OMP.exe && ./gaussian_OMP.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         1.00    1.00    -1.00   4.00
         2.00    2.00    -1.00   7.00
         0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         2.00    1.00    0.00    3.00

The resulting Augmented Matrix M:
         1.00    1.00    -1.00   4.00    2.00
         2.00    2.00    -1.00   7.00    1.00
         0.00    0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00   3.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    1.00    0.00    3.00    -1.00
         0.00    0.00    1.00    -1.00   -3.00
         0.00    0.00    0.00    0.00    0.00
         0.00    0.00    0.00    0.00    0.00

The linear system has infinitely many solutions, because rank(A)=2 == rank(M)=2 < N_var=4.
Run time for Gaussian elimination openMP: 27.256966 milliseconds
[sgyzh180@viz02[barkla] compare]$ mpiicc -O0 gaussian_MPI.c gaussianLib.c -o gaussian.exe && mpirun -np 2 ./gaussian.exe 2
size: N_cons, N_var = 4, 4

Coefficient Matrix A:
         1.00    1.00    -1.00   4.00
         2.00    2.00    -1.00   7.00
         0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00
Variables x.T:
         x1      x2      x3      x4
Constants b.T:
         2.00    1.00    0.00    3.00

The resulting Augmented Matrix M:
         1.00    1.00    -1.00   4.00    2.00
         2.00    2.00    -1.00   7.00    1.00
         0.00    0.00    0.00    0.00    0.00
         3.00    3.00    -2.00   11.00   3.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    1.00    0.00    3.00    -1.00
         0.00    0.00    1.00    -1.00   -3.00
         0.00    0.00    0.00    0.00    0.00
         0.00    0.00    0.00    0.00    0.00

The linear system has infinitely many solutions, because rank(A)=2 == rank(M)=2 < N_var=4.
Run time for Gaussian elimination of by MPI: 0.418715 milliseconds
[sgyzh180@viz02[barkla] compare]$
```

3. Test case 3 infeasible by Serial – OMP with 2 threads – MPI with 3 threads



```
[sgyzh180@viz02[barkla] compare]$ clear
[sgyzh180@viz02[barkla] compare]$ icc -O0 gaussian_serial.c gaussianLib.c -o gaussian_serial.exe && ./gaussian_serial.exe 2
size: N_cons, N_var = 3, 3

Coefficient Matrix A:
         3.00    6.00    6.00
         3.00    -6.00   -3.00
         3.00    -2.00   0.00
Variables x.T:
         x1      x2      x3
Constants b.T:
         5.00    2.00    1.00

The resulting Augmented Matrix M:
         3.00    6.00    6.00    5.00
         3.00    -6.00   -3.00   2.00
         3.00    -2.00   0.00    1.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.50    0.00
         0.00    1.00    0.75    0.00
         0.00    0.00    0.00    1.00

The linear system is infeasible, since rank(A)=2 != rank(M)=3.
Run time for Gaussian elimination by serial algorithm: 0.000604 milliseconds
[sgyzh180@viz02[barkla] compare]$ icc -qopenmp -O0 gaussian_OMP.c gaussianLib.c -o gaussian_OMP.exe && ./gaussian_OMP.exe 2
size: N_cons, N_var = 3, 3

Coefficient Matrix A:
         3.00    6.00    6.00
         3.00    -6.00   -3.00
         3.00    -2.00   0.00
Variables x.T:
         x1      x2      x3
Constants b.T:
         5.00    2.00    1.00

The resulting Augmented Matrix M:
         3.00    6.00    6.00    5.00
         3.00    -6.00   -3.00   2.00
         3.00    -2.00   0.00    1.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.50    0.00
         0.00    1.00    0.75    0.00
         0.00    0.00    0.00    1.00

The linear system is infeasible, since rank(A)=2 != rank(M)=3.
Run time for Gaussian elimination openMP: 27.999878 milliseconds
[sgyzh180@viz02[barkla] compare]$ mpiicc -O0 gaussian_MPI.c gaussianLib.c -o gaussian.exe && mpirun -np 2 ./gaussian.exe 2
!! N_cons 3 is indivisible by comm_size 2 , please change the number of processors !!
[sgyzh180@viz02[barkla] compare]$ mpiicc -O0 gaussian_MPI.c gaussianLib.c -o gaussian.exe && mpirun -np 3 ./gaussian.exe 2
size: N_cons, N_var = 3, 3

Coefficient Matrix A:
         3.00    6.00    6.00
         3.00    -6.00   -3.00
         3.00    -2.00   0.00
Variables x.T:
         x1      x2      x3
Constants b.T:
         5.00    2.00    1.00

The resulting Augmented Matrix M:
         3.00    6.00    6.00    5.00
         3.00    -6.00   -3.00   2.00
         3.00    -2.00   0.00    1.00

Gaussian Elimination to the reduced row-echolen form:
         1.00    0.00    0.50    0.00
         0.00    1.00    0.75    0.00
         0.00    0.00    0.00    1.00

The linear system is infeasible, since rank(A)=2 != rank(M)=3.
Run time for Gaussian elimination of by MPI: 0.499777 milliseconds
[sgyzh180@viz02[barkla] compare]$
```