Assignment Coversheet – Individual Assignment

Please fill in your details below. Use one form for each assignment.

Personal Details:

Family Name	Given Name (s)	Student Number (SID)	Unikey	Signature
Gao	**Hongkai**	**520311802**	hgao7738	Gao Hongkai

Assignment Details:

Assignment Title	**Assignment 2 MPI programming**		
Assignment Number	**2**		
Unit of Study Tutor	**James Phung**		
Tutorial ID			
Due Date	**2024.05.29**	Submission Date	**2024.05.24**

Declaration:

1. I understand that all forms of plagiarism and unauthorised collusion are regarded as academic dishonesty by the university, resulting in penalties including failure of the unit of study and possible disciplinary action.
2. I have completed the **Academic Honesty Education Module** on Canvas.
3. I understand that failure to comply with the Academic Dishonesty and Plagiarism in Coursework Policy can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the ***University of Sydney By-Law 1999*** (as amended).
4. This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that it is not my own by acknowledging the source of that part or those parts of the work.
5. The assessment has not been submitted previously for assessment in this or any other unit, or another institution.
6. I acknowledge that the assessor of this assignment may, for the purpose of assessing this assignment may:
    a. Reproduce this assignment and provide a copy to another member of the school; and/or
    b. Use similarity detection software (which may then retain a copy of the assignment on its database for the purpose of future plagiarism checking).
7. I have retained a duplicate copy of the assignment.

Please type in your name here to acknowledge this declaration:	Gao Hongkai

1. Problem Definition and Requirements

1.1. Problem Statement:

Gaussian elimination with partial pivoting is a numerical method used to solve systems of linear equations [1]. It addresses the issue of numerical instability caused by small or zero coefficients in the coefficient matrix. Partial pivoting involves selecting the element with the largest absolute value in each column as the pivot element before performing elimination [2]. This helps minimize rounding errors and enhances computational stability.

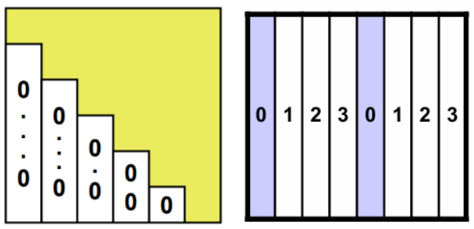


Figure 1.1 Left shows Result matrix. Right is Column block cyclic.

1.2. Problem Requirements:

In distributed memory machines, data need to be partitioned and distributed to processes. Column block cyclic partitioning is a data distribution scheme commonly used in parallel linear algebra libraries, such as ScaLAPACK, for efficient load balancing and communication among processes [3]. Both programs need to conduct a self-check for correctness. Figure 1.1 shows the outcoming of Gaussian elimination and how column block cyclic works.

1.2.1. Task1

- Implement Gaussian elimination with partial pivoting using MPI, without loop unrolling.
- Adopt column block cyclic partitioning with variable block size b.
- Ask for matrix size N and block size b as input parameters.

1.2.2. Task2

- Implement Gaussian elimination with partial pivoting using MPI, with loop unrolling.
- With fixed block size b=8 and loop unrolling factor of 4.
- Ask for matrix size N as an input parameter.

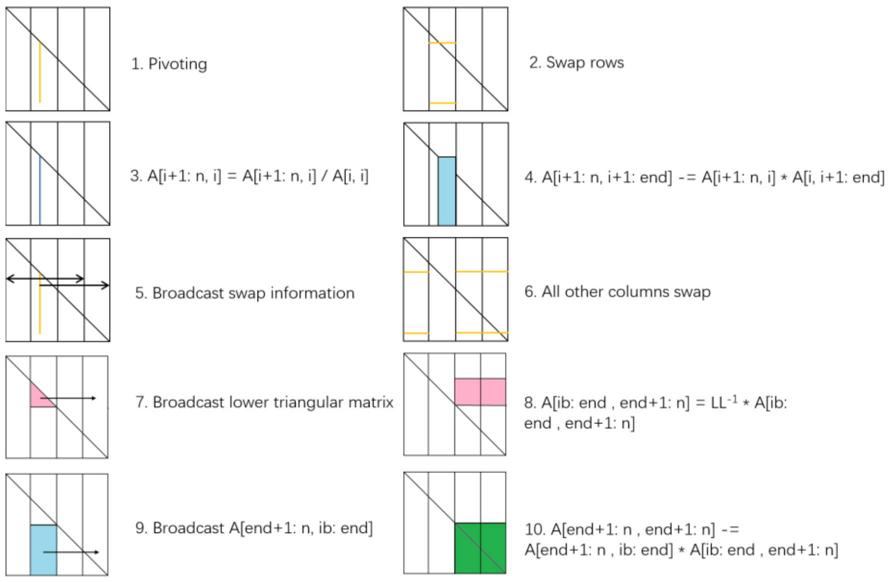
2. Algorithm Design and Implementation

In this assignment, both task1 and task2 follow the same algorithm BLAS3 Gaussian elimination with partial pivoting. I will describe the major part in task1 and some difference in task2.

2.1. Task1

2.1.1. Task1 Algorithm

Figure 2.1.1 shows for distributed memory system, there is a modification version of BLAS3 Gaussian elimination with partial pivoting.



1. Pivoting
2. Swap rows
3. A[i+1: n, i] = A[i+1: n, i] / A[i, i]
4. A[i+1: n, i+1: end] -= A[i+1: n, i] * A[i, i+1: end]
5. Broadcast swap information
6. All other columns swap
7. Broadcast lower triangular matrix
8. A[ib: end , end+1: n] = LL^{-1} * A[ib: end , end+1: n]
9. Broadcast A[end+1: n, ib: end]
10. A[end+1: n , end+1: n] -= A[end+1: n , ib: end] * A[ib: end , end+1: n]

Figure 2.1.1 BLAS3 with column block cyclic, when 2nd block is working.

Each processor (including 0) has local AK and AW. No.0 processor initialize matrix A, then partition the data into each processor's AK with load balancing.

Brown Part 1. 2.

Iterate in column blocks of size n x b, starting from the top-left corner.

For each block, perform partial pivoting:

Find the element with the largest abstract value in the current column, only in the strict lower triangular matrix.

If the row with the maximum element is not the current row, swap the entire rows.

Store the swap information.

Blue Part 3. 4.

Update the elements below in the current block

Divide the elements below the diagonal in the current column by the diagonal.

$$AK[i+1:n,i] = AK[i+1:n,i]\ /\ AK[i,i]$$

Update the elements in the following columns of the current block.

$$AK[i+1:n,i+1:end] -= AK[i+1:n,i] * AK[i,i+1:end]$$

Others Brown Part 5. 6.,

Broadcast swap information, size equals to block size to all other processors.

Swap rows in all other processors.

Pink Part 7.8.

Copy lower triangular matrix from *AK* to *AW*, then broadcast AW to all other processors.

If *myid* <= *working processor*:

Column index *end* start from where working processor's blue part ends.

Else:

Column index *end* start from where working processor's blue part starts.

In-place inversion of the LL matrix using the Gauss-Jordan elimination method

$$LL[i,j] = LL[i,j]/LL[i,i]\ LL[k,j] = LL[k,j] - LL[k,i] * LL[i,j]$$

Calculate and update the result.

$$AK[ib:end,end+1:n] = LL^{-1}[ib:end,ib:end] * AK[ib:end,end+1:n]$$

Green Part 9. 10.

Copy column block matrix from *AK* to *AW*, then broadcast *AW* to all other processors.

Calculate and update green part of AK using matrix multiplication

$$AK[end+1:n,end+1:n] -= AW[end+1:n,ib:end] * AK[ib:end,end+1:n]$$

No.0 processor receive AK from all other processors and update value in A.

Figure 2.1.2 shows the task dependency graph during elimination. There are 2 types of processors: Working (Blue part in process) and Other (Only Pink and Green).

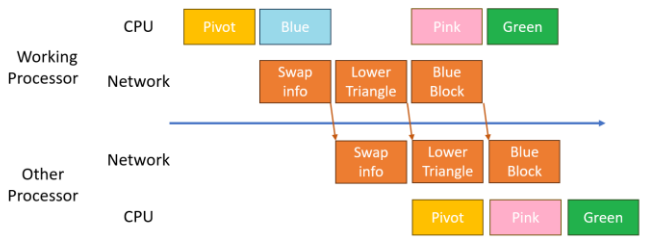
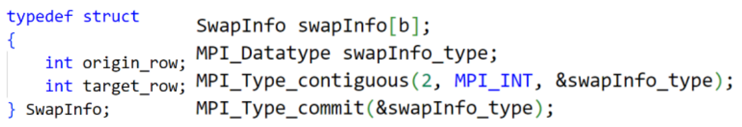


Figure 2.1.2 Task dependency graph for GEPP with column block cyclic.

2.1.2. Task1 Implementation

Create Struct to record swap information, contiguous datatype in MPI for broadcast. Figure 2.1.3 shows how swap information is stored and communicated during the elimination.

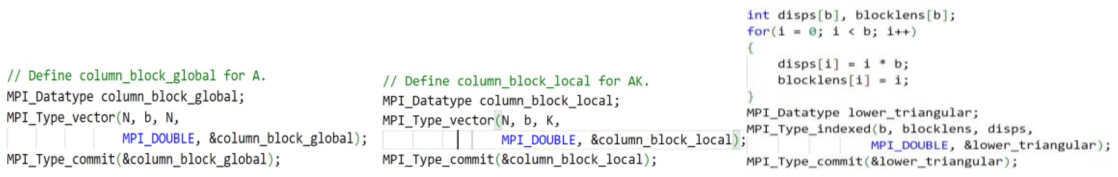


```
typedef struct
{
    int origin_row;
    int target_row;
} SwapInfo;
```

```
SwapInfo swapInfo[b];
MPI_Datatype swapInfo_type;
MPI_Type_contiguous(2, MPI_INT, &swapInfo_type);
MPI_Type_commit(&swapInfo_type);
```

Figure 2.1.3 Key-Value Pair. Store [origin, target] in struct

The most important part in MPI programming is communication. Figure 2.1.4 shows (a) column block for whole matrix A, (b) column block for local matrix AK and (c) lower triangular matrix with indexed datatype for communicating matrix AW.



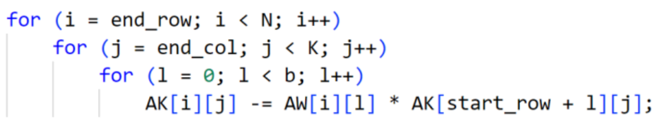
```
// Define column_block_global for A.
MPI_Datatype column_block_global;
MPI_Type_vector(N, b, N,
                MPI_DOUBLE, &column_block_global);
MPI_Type_commit(&column_block_global);
```

```
// Define column_block_local for AK.
MPI_Datatype column_block_local;
MPI_Type_vector(N, b, K,
                MPI_DOUBLE, &column_block_local);
MPI_Type_commit(&column_block_local);
```

```
int disps[b], blocklens[b];
for(i = 0; i < b; i++)
{
    disps[i] = i * b;
    blocklens[i] = i;
}
MPI_Datatype lower_triangular;
MPI_Type_indexed(b, blocklens, disps,
                 MPI_DOUBLE, &lower_triangular);
MPI_Type_commit(&lower_triangular);
```

Figure 2.1.4 MPI datatype declaration.

Instead of actually calculating inverse of LL, I use methods provided by lecturer in eliminating pink part to reduce memory usage and numbers of operations. Figure 2.1.5 shows directly update Pink Part using LL without inversion and extra matrix declaration.



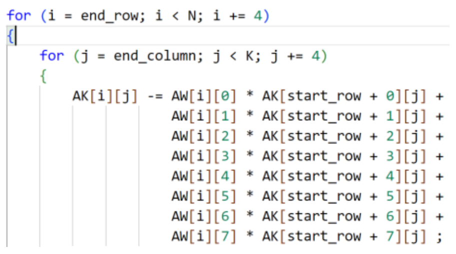
```
for (i = end_row; i < N; i++)
    for (j = end_col; j < K; j++)
        for (l = 0; l < b; l++)
            AK[i][j] -= AW[i][l] * AK[start_row + l][j];
```

Figure 2.1.5 No need to inverse LL.

2.2. Task2

Overall structure is the same as task1. Additionally, with the experience from Assignment 1, most time is spent on green part. So, here I only focus on unrolling green part to get performance boost. With fixed block size = 8 and unrolling factor = 4, *ib : end* is fixed to 8.

$$AK[end+1{:}n, end+1{:}n] -= AW[end+1{:}n, ib{:}end] * AK[ib{:}end, end+1{:}n]$$

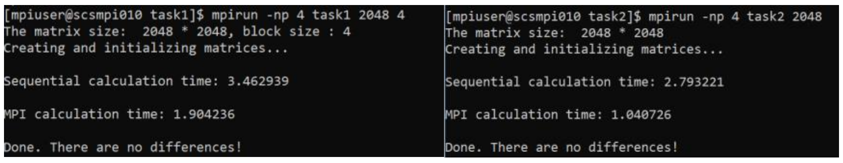


```
for (i = end_row; i < N; i += 4)
{
    for (j = end_column; j < K; j += 4)
    {
        AK[i][j] -= AW[i][0] * AK[start_row + 0][j] +
                    AW[i][1] * AK[start_row + 1][j] +
                    AW[i][2] * AK[start_row + 2][j] +
                    AW[i][3] * AK[start_row + 3][j] +
                    AW[i][4] * AK[start_row + 4][j] +
                    AW[i][5] * AK[start_row + 5][j] +
                    AW[i][6] * AK[start_row + 6][j] +
                    AW[i][7] * AK[start_row + 7][j] ;
```

Figure 2.2.1 Example of Unrolling.

Figure 2.2.1 shows one of 16 unrolling lines for Green part calculation. Each line has 8 items.

3. Testing and performance evaluation

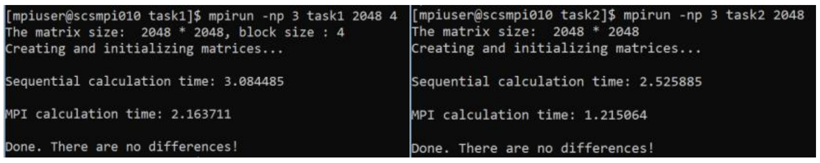


```
[mpiuser@scsmpi010 task1]$ mpirun -np 4 task1 2048 4
The matrix size:  2048 * 2048, block size : 4
Creating and initializing matrices...

Sequential calculation time: 3.462939

MPI calculation time: 1.904236

Done. There are no differences!
```

```
[mpiuser@scsmpi010 task2]$ mpirun -np 4 task2 2048
The matrix size:  2048 * 2048
Creating and initializing matrices...

Sequential calculation time: 2.793221

MPI calculation time: 1.040726

Done. There are no differences!
```

Figure 3.1 Test in Azure Virtual Machine, left for task1 and right for task2

Figure 3.1 and 3.2 shows that program can be compiled and run in Azure virtual machine. 3.1 for normal case np = 4, N = 2048, b = 4 and 3.2 for edge case np = 3, N = 2048



```
[mpiuser@scsmpi010 task1]$ mpirun -np 3 task1 2048 4
The matrix size:  2048 * 2048, block size : 4
Creating and initializing matrices...

Sequential calculation time: 3.084485

MPI calculation time: 2.163711

Done. There are no differences!
```

```
[mpiuser@scsmpi010 task2]$ mpirun -np 3 task2 2048
The matrix size:  2048 * 2048
Creating and initializing matrices...

Sequential calculation time: 2.525885

MPI calculation time: 1.215064

Done. There are no differences!
```

Figure 3.2 Edge case test

Table 3.1 shows task1 performance evaluation. Block size is set to $2^{\log_{10} N}$.Red means actual efficiency reduction.

MATRIX SIZE(N)	THREAD NUM(T)	BLOCK SIZE(B)	SEQUENTIAL (S)	MPI(S)	SPEEDUP
128	**2**	**4**	**0.000699**	**0.008169**	**1168.67%**
	4	4	0.000699	0.01289	1844.06%
2048	2	8	2.616234	3.10984	118.87%
	4	**8**	**2.616234**	**1.914792**	**73.19%**
10000	2	16	582.760779	297.706505	51.09%
	4	**16**	**582.760779**	**156.788483**	**26.90%**

Table 3.1 Task1 performance evaluation

Table 3.2 shows task2 performance evaluation. In task2 block size is set to 8 and unrolling factor to 4. Red means actual efficiency reduction.

MATRIX SIZE(N)	THREAD NUMS (T)	SEQUENTIAL (S)	MPI BLOCK + UNROLL(S)	SPEEDUP
128	**2**	**0.000699**	**0.004647**	**664.81%**
	3	0.000699	0.005818	832.33%
2048	2	2.616234	1.726792	66.00%
	3	**2.616234**	**0.946575**	**36.18%**
	4	2.616234	1.182015	45.18%
10000	2	582.760779	196.3754086	33.70%
	3	582.760779	144.1851997	24.74%
	4	**582.760779**	**109.711532**	**18.83%**

Table 3.2 Task2 performance evaluation

4. Discussion

4.1. Correctness check

Task1 and Task2 both pass the correctness check provided in gepp_3.c

4.2. Task1 result implication

The MPI implementation shows the effectiveness of parallel processing for larger matrix sizes with significant speedup ratios. However, for small problem size, the parallel overhead outweighs the benefit. The speed up ratio is not ideal, pointing out the presence of factors such as **communication overhead** and **serial portions** of the code that limit the overall parallel efficiency. Moreover, due to the limitation of time, I don't experiment on different block size, instead setting to $2^{\log_{10} N}$. Allover, result highlights the importance of considering problem size, the number of processes, and the trade-offs involved in applying MPI techniques.

4.3. Task2 result implication

For small matrices (N=128), the parallel overhead remains significant. As the matrix size increases to medium (N=2048) and large (N=10000), the computational workload becomes more dominant, explaining the use of more threads to distribute the work and achieve better performance. Compared to the previous results without optimizations, blocking and loop unrolling boost the parallel performance by improving **cache utilization** and reducing l**oop overhead.**

5. Known issues in the program

Use INTEGER for input of N. Larger N will exceed the range of INTERGER.

Due to the virtual machine usage, internet connection and memory allocation and maximum 4 cores in VM virtual box, more extensive experiments cannot be conducted.

6. Manual

For task1.c	Compile: mpicc -o task1 task1.c -O3 Run: mpirun -np 2 task1 2048 8
For task2.c	Compile: mpicc -o task2 task2.c -O3 Run: mpirun -np 2 task2 2048

7. Reference

[1] Randall, T. J. (1985). Matrix computations, by Gene H. Golub and Charles F. Van Loan. Pp 476. 1983. ISBN 0-8018-3010-9 (Johns Hopkins University Press). Mathematical Gazette, 69(448), 152–152. https://doi.org/10.2307/3616959

[2] Strawderman, R. L. (1999). Accuracy and Stability of Numerical Algorithms [Review of Accuracy and Stability of Numerical Algorithms]. Journal of the American Statistical Association, 94(445), 349–350. American Statistical Association. https://doi.org/10.2307/2669725

[3] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., & Whale, R. C. (1997). ScaLAPACK users' guide. Society for Industrial and Applied Mathematics.