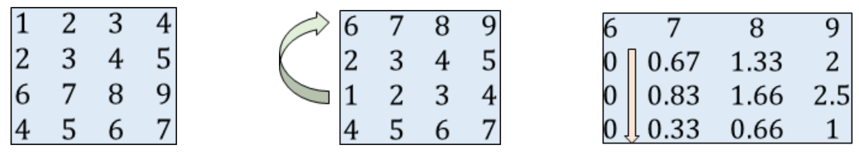
COMP5426 Assignment 2

Sicheng Liu
SID: 520096356
Unikey: sliu3452

1 Problem Definition and Requirements

1.1 Gaussian Elimination with Pivoting

Gaussian elimination with pivoting is a computationally efficient method of solving linear equations. It solves linear equations by performing row transformations on the matrix which simplifies the linear equation. Pivoting aims to find the largest number in the current column improving the stability of the Gaussian elimination. Specifically, the whole algorithm starts from the first column and ends with the last column. Pivoting is firstly applied to find the largest number in the first column and move the entire row to the first row. We call the largest number in the corresponding column pivot. Then we eliminate all elements in the first column by subtracting the product of the corresponding element in pivot row with $m_{j,i}$ in each row, where $m_{j,i} = \frac{a_{j,i}}{a_{i,i}}$ is the quotient of the element in the same column as the pivot in the current row and the pivot element.is the quotient of the element in the same column as the pivot and the pivot element. Figure 1 shows the steps to eliminate the elements in the first column. These steps will repeat to convert the input matrix to the echelon matrix.



(a) Original matrix (b) Pivoting (c) Gaussian elimination

Figure 1: The process of Gaussian elimination with pivoting. (a) The value in the original matrix. (b) The first row is swapped with the third row in the pivoting stage. (c) Apply Gaussian elimination to make all elements in the first column except pivot to zero, and update the remaining elements.

1.2 Tasks

In this assignment, we are asked to use blocking to improve the efficiency of Gaussian elimination. The blocking algorithm needed to be implemented on the distributed memory platform. There are two sub-tasks.

Blocking with changeable block size. We need to implement blocking-based Gaussian elimination with a user-specified block size on the distributed memory machine.

Blocking with Loop Unrolling. We need to use both loop unrolling technique and blocking to improve the Gaussian elimination efficiency on the distributed memory machine.

2 Algorithm Design and Implementation

2.1 Blocking on Distributed Memory Machine

The blocking algorithm aims to divide the matrix into several small blocks. These blocks are small enough to be stored in the cache. Therefore, the data in the cache can be used effectively and computing efficiency can be improved based on this. The whole algorithm contains 3 steps including matrix partition, Gaussian elimination and result mergence.

2.1.1 Partition

As our task requires running on the distributed memory machine, different blocks are stored in different machines. We adopt column 1D block cyclic partitioning which can trade load balance and BLAS3 performance [1]. The original matrix is partitioned into several blocks, and each processor stores blocks cyclically. As shown in Figure 2, if a matrix contains 7 blocks in total, processor 0, 1, 2 will store 2 blocks while processor 3 will store 1 block. The black number in figure is the block index and red number is the processor number.

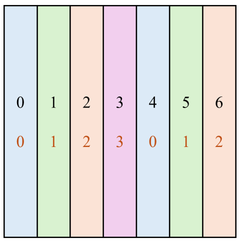


Figure 2: 1D column block cyclic layout

2.1.2 Gaussian Elimination

As shown in Figure 3, the whole matrix A with size M can be divided into four parts which are coloured white, blue, pink, green respectively. The white part is already updated thus we only need to focus on the rest parts.

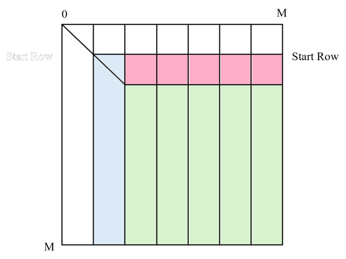


Figure 3: Gaussian elimination with blocking

As blocks are stored in different machine, processors need to communicate with each other to exchange data during Gaussian elimination. Taking the current block on processor 0 as an example, Figure 4 shows the overall calculation process for the current block.

As there is no shared value in the distributed memory machine, in practice, we calculate the $StartRow$ through the block index to reduce the communication.

$$S_R = StartRow = k * blocksize \tag{1}$$

where k is the index of the blocks. We also calculate the $StartColumn$ since the $StartColumn$ is not equal to the $StartRow$ when calculating on distributed memory machine.

$$S_C = StartColumn = (k\%n) * blocksize \tag{2}$$

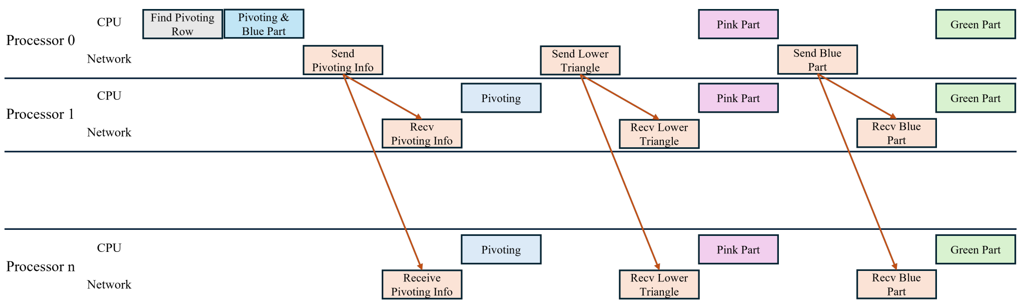


Figure 4: Overall computation sequence (when the current block is on processor 0)

where n is the total number of processors. To simplify calculation, the end column E_C and end row E_R are calculated by:

$$E_C = S_C + blocksize \tag{3}$$
$$E_R = E_R + blocksize \tag{4}$$

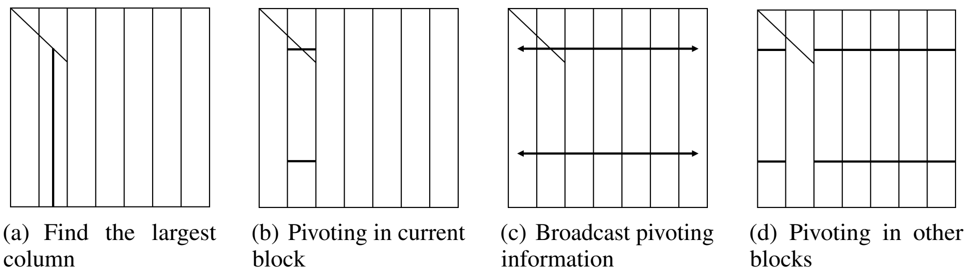
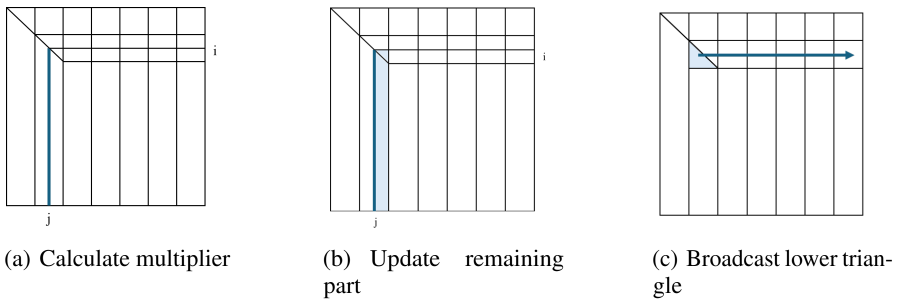

(a) Find the largest column (b) Pivoting in current block (c) Broadcast pivoting information (d) Pivoting in other blocks

Figure 5: Pivoting

Pivoting. is required before elimination. It aims to find the largest number below the current row as the pivoting row and swap the entire pivoting row with the current row. Figure 5 shows the process of pivoting. Specifically, we first find the largest number in the current column below the $StartRow$. Then we swap the current row with the pivoting row. We use a structure array of size $blocksize$ to record line-swapping information. The structure contains two values, one is the current row and the other is the pivoting row. Then, the structure array is broadcast to every processor. Other processors swap rows in sequence based on the structure array information.



(a) Calculate multiplier (b) Update remaining part (c) Broadcast lower triangle

Figure 6: Updating blue part

Blue Part represent the $A[S_R : M, S_C : E_C]$. Figure 6 show the updating process. To update this part, we first calculate the multiplier by dividing all elements of the current column by the pivot through Equation 5.

$$A[i+1:N,j] = A[i+1:N,j]/A[i,j] \quad \text{s.t. } i \in [S_R, S_R + blocksize] \tag{5}$$

where i is current row and j is current column. The multipliers are stored in $A[i+1:N,j]$. They are used to update the remaining part $A[i+1:N,j:E_C]$ by Equation .

$$A[i+1:N,j+1:E_C] = A[i+1:N,j+1:E_C] - A[i+1:N,j] * A[i,j+1:E_C] \tag{6}$$

After updating current block $A[S_R : M, S_C : E_C]$, the processor need to broadcast lower triangle information to help the updating for pink part.

Pink Part is the matrix on the right side of the lower triangle in the original matrix. However, as blocks are stored separately, we need to determine which part in current processor are inside the pink part. The start column ($S_{C_{pink}}$) is calculated by:

$$S_{C_{pink}} = \begin{cases} E_C & \text{if } i \leq (k\%n) \\ S_C & \text{if } i \geq (k\%n) \end{cases} \tag{7}$$

where i is the id of current processor. On each processor, the pink part can be represent identically as $A[S_R : E_R, S_{C_{pink}} : M]$.

The lower triangle is denoted as LL. We first inverse it and then update $A[S_R : E_R, S_{C_{pink}} : M]$ through:

$$LL = LL^{-1} \tag{8}$$

$$A[S_R : E_R, S_{C_{pink}} : M] = LL * A[S_R : E_R, S_{C_{pink}} : M] \tag{9}$$

In practice, in order to facilitate code implementation, we follow the [1] to update pink part which do not require matrix inversion.

Green Part is the matrix in the lower right corner. Similarly, its start column is indeterminate in different processors. However, they are equal to the $S_{C_{pink}}$. Therefore, we denote the start column in green part as $S_{C_{green}} = S_{C_{pink}}$.

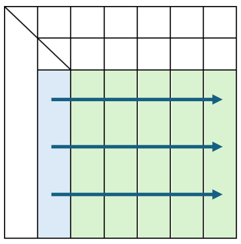


Figure 7: Updating green part

To update this part, the information from blue part is needed. Therefore, as shown in Figure 7 we broadcast the blue part. Then the green part can be updated through Equation 10.

$$A[E_R : M, S_{C_{green}} : M] = A[E_R : M, S_{C_{green}} : M] - A[E_R : M, S_C : E_C] * A[S_R : E_R, S_{C_{green}} : M] \tag{10}$$

2.2 Changeable Block Size

To deal with the changeable block size, the lower triangle is stored in a matrix with the size of $blocksize - 1$. The method to update the pink part is shown in Algorithm 1. By adding a new loop, the inner loop can calculate one element in the pink part.

Algorithm 1 Updating pink part with changeable block size

```
**Input:** Matrix $AK$, Lower triangular matrix $LL$, Start row $start\_row\_pink$, End column $end\_column$, Block size $block\_size$, Number of columns $K$
for $j = end\_column$ to $K - 1$ do
    for $i = 0$ to $block\_size - 2$ do
        for $l = 0$ to $block\_size - 2$ do
            $AK[i + start\_row\_pink + 1][j] -= LL[i][l] \cdot AK[start\_row\_pink + l][j]$
        end for
    end for
end for
```

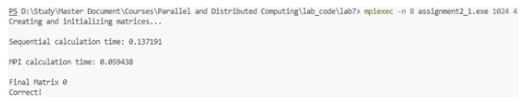
2.3 Loop Unrolling

Loop unrolling can expand the loop making the program execute several steps in one cycle. In this assignment, the unrolling factor is set to 4, which means the program will execute 4 steps in one cycle. I applied loop unrolling in updating the green part mentioned before, which requires largest computation time in the whole algorithm. We unroll two loops. The four columns and four rows will calculate together. Since we use '-O3' to compile the code, we do not explicitly use register in our code.

3 Testing and Performance Evaluation

3.1 Correctness Testing

A self-check function is implemented to check the correctness of the algorithm. We compare the results of blocked Gaussian elimination and the results of the original Gaussian elimination. The result is shown in Figure 8. We test our algorithm with different block sizes, the number of process and the matrix size.



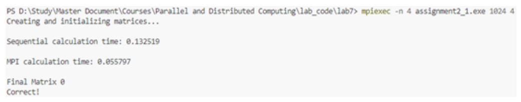
```
PS D:\Study\Master Document\Courses\Parallel and Distributed Computing\lab_code\lab7> mpiexec -n 8 assignment2_1.exe 1024 4
Creating and initializing matrices...

Sequential calculation time: 0.137191

MPI calculation time: 0.059438

Final Matrix 0
Correct!
```

(a) Result of the matrix size of 1024, block size of 4, 8 processor



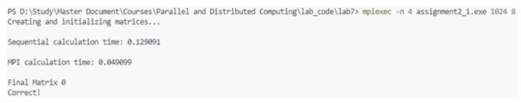
```
PS D:\Study\Master Document\Courses\Parallel and Distributed Computing\lab_code\lab7> mpiexec -n 4 assignment2_1.exe 1024 4
Creating and initializing matrices...

Sequential calculation time: 0.132519

MPI calculation time: 0.055797

Final Matrix 0
Correct!
```

(b) Result of the matrix size of 1024, block size of 4, 4 processor



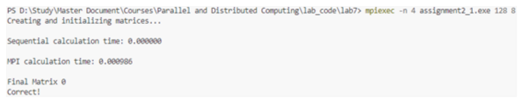
```
PS D:\Study\Master Document\Courses\Parallel and Distributed Computing\lab_code\lab7> mpiexec -n 4 assignment2_1.exe 1024 8
Creating and initializing matrices...

Sequential calculation time: 0.129091

MPI calculation time: 0.049099

Final Matrix 0
Correct!
```

(c) Result of the matrix size of 1024, block size of 8, 4 processor



```
PS D:\Study\Master Document\Courses\Parallel and Distributed Computing\lab_code\lab7> mpiexec -n 4 assignment2_1.exe 128 8
Creating and initializing matrices...

Sequential calculation time: 0.000000

MPI calculation time: 0.000986

Final Matrix 0
Correct!
```

(d) Result of the matrix size of 128, block size of 8, 4 processor

Figure 8: Correctness testing

3.2 Comparison between methods

We first compare the algorithm between different methods including original, blocking and blocking with loop unrolling Gaussian elimination. As loop unrolling requires a block size of 8, we set the block size to 8 during the experiment. We use 4 processors for the paralyzed algorithm. The experiment results are shown in Table 1.

Table 1: Performance comparison of matrix operations

Matrix Size	Original	Blocking		Blocking+Unrolling	
	time	time	speed up	time	speed up
1024	0.1369	0.0546	2.51	0.0297	5.61
5120	43.3988	6.15	7.06	4.2	10.33
10240	262.705	49.533	5.303	31.827	8.25

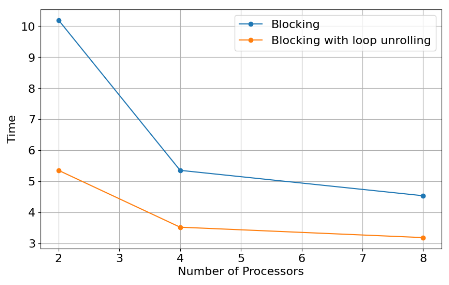
3.3 Parameter Tuning

We also test the performance with different numbers of processors and different block sizes. Since adjusting the block size will make loop unrolling unavailable, we only tested the impact of different block sizes on the blocking algorithm. During parameter tuning, the matrix size is set to 5120 and the block size is 8. The algorithm performance with different number of processors are shown in Figure 9(a). The blue line shows the performance of the blocking algorithm and orange line shows the result of blocking with loop unrolling. We set the processor number to 4 when testing the algorithm performance with different block sizes. Figure 9(b) shows the experiment result.

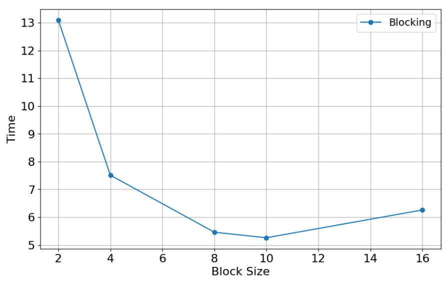
4 Discussion

4.1 Method Comparision

From Table 1, we can see that algorithms using blocking and unrolling can have the best performance. When the matrix size is 5120, the calculation time is one-tenth of the original method, which is the time when the performance improvement percentage is the largest. In addition, using only blocking can



(a) Result of different number of processors



(b) Result of different block size

Figure 9: Experiment results

also achieve good performance improvements. When the matrix size is 5120, the calculation time is one-seventh of the original method. It can be seen that as the matrix continues to increase, the performance improvement percentage of both optimization methods decreases. In general, the two optimization algorithms have been successful, can greatly accelerate the operation of Gaussian elimination.

4.2 Parameter Tuning

It can be seen from Figure 9(a) that as the number of processors increases, the algorithm time consumption decreases. But when the number of processors increases from 4 to 8, the reduction in algorithm time is not so obvious. This may be caused by two reasons. The first is hardware limitations. Since the CPU in the test environment has a performance core and an energy efficiency core, the calculation speed of the energy efficiency core will be slower than that of the performance core. So when the processor is used more, energy-efficient cores are used, prolonging the calculation time. Second, as the number of processors increases, the communication between processors also increases, resulting in increased communication time. This also results in extended computation time.

When changing the block size, we found that the algorithm has the best performance when the block size is 10 as shown in Figure 9(b). This is possible because when the block size is 10, the computational burden of each part is evenly distributed, thereby increasing the degree of algorithm parallelism.

5 Known Issues in the Program

There are several known issues in the program and the experiment.

- We only conducted one test instead of averaging multiple tests. Since the CPU's work efficiency will vary at different times, the results obtained from the experiment may fluctuate ($\pm 10\%$).
- Due to time limitations, I do not explore algorithm performance with the larger matrix. Not sure whether the optimized algorithm will still perform well on larger matrices.

6 Manual

Here are the instructions to run the program:

use **Makefile**:

- Compile: run "mingw32-make".
- Run task 1: run "mpiexec -n 4 assignment2_1.exe 1024 4".
- Run task 2: run "mpiexec -n 4 assignment2_2.exe 1024".

Attention, you may need to edit the makefile to use the correct path for MPI.

References

[1] Bingbing Zhou. Parallel algorithm design for distributed-memory machines. *COMP5426 Parallel and Distributed Computing, The University of Sydney*, 2024.