Ain Shams University

Faculty of Engineering

Computer Engineering and Software Systems Program

Fall 2025

CSE355: Parallel and Distributed Algorithms

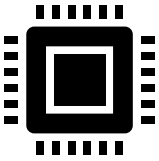
Assignment3: Prepare a report showing how to fix the problems in matrix multiplication to allow for using all matrix chunks to be processed and not only the first 3.

Presented by:

YOUSSEF MOUSTAFA ELSAYED 22P0047

Assignment 3 Report

Prepare a report showing how to fix the problems in matrix multiplication to allow for using all matrix chunks to be processed and not only the first 3.



CSE355: Parallel and Distributed Algorithms

Presented to:

Eng. Hamdy Osama

Presented by:

Youssef Moustafa Elsayed 22P0047

1. Introduction

Parallel computing enables large computations to be divided among multiple processors, reducing execution time and improving performance. Message Passing Interface (MPI) is one of the most widely used standards for writing parallel programs, especially when working with distributed memory systems. In this experiment, we implement a parallel matrix–vector multiplication using MPI. The goal is to divide the matrix rows evenly across all processes, allow each process to compute a portion of the result independently, and then gather the partial results into a final output vector. This demonstrates core MPI concepts such as broadcasting shared data, scattering distributed work, local computation, and gathering results, while highlighting the importance of correct data distribution in parallel algorithms.

.

2. Problem Definition

The program performs parallel matrix–vector multiplication using MPI.

$$\textbf{vector\_out} = \textbf{matrix} \times \textbf{vector\_in}$$

The main idea is to divide the computation among several processes so that each process handles a portion of the matrix independently.
First, one process (rank 0) generates a matrix and a vector. The vector is then shared with all processes, since every process needs it for its computation. The matrix, however, is divided into equal blocks of rows, and each block is sent to a different process.
Each process computes the dot products between its assigned matrix rows and the input vector. These partial results represent the corresponding entries of the final output vector. After finishing their local computations, all processes send their partial results back to the root process, which combines them into the final output vector.

2.1. Code

6.4-still-broken-matmult-mpi.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define ROWS 9
#define COLS 5

// This is just to visualize the problem to you my dear students, you may ignore it
void print_matrix_and_vectors(int*, int*, int*);

int main() {

    MPI_Init(NULL, NULL);
    int world_size, my_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // We want to do vector_out = matrix * vector_in
    int* vector_in = (int*) malloc(COLS * sizeof(int)); // Shared, need to malloc for all
    // XXX: We can't use 2D arrays with MPI! MPI functions expect contiguous data
    // int** matrix;
    int* matrix; // Unique to rank #0 process, will only allocate memory for it
    int* vector_out; // Unique to rank #0 process, will only allocate memory for it
```

- The root process Allocates a flattened 1D array for the matrix instead of 2D one, then make nested for loops to fill matrix with random small numbers.
- Creates vector_in and fills it with random values.
- Allocates vector_out (initially all zeros).

```
25      // Rank #0 shall randoml                 vectors 1 and 2 and print them
26      if (my_rank == 0) {
27
28          // XXX: We shall                    ne the matrix as 1-dimensional array
29          // matrix = (int                    loc(ROWS * sizeof(int*));
30          // for (int i =                     < ROWS; i++) {
31          //     matrix                       (int*) malloc(COLS * sizeof(int));
32          //     for (                        = 0; j < COLS; j++) {
33          //                                  rix[i][j] = rand() % 5;
34          //     }
35          // }
36          matrix = (int*) malloc(ROWS * COLS * sizeof(int));
37          for (int i = 0; i < ROWS; i++) {
38              for (int j = 0; j < COLS; j++) {
39                  matrix[COLS * i + j] = rand() % 5;
40              }
41          }
42
43          for (int i = 0; i < COLS; i++) vector_in[i] = rand() % 4;
44
45          vector_out = (int*) calloc(ROWS, sizeof(int));
46
47      }
```

Use MPI_Bcast to Sends vector_in from process 0 to all other processes and now every process has the same vector needed for multiplication.

```
49      // Since they all use vector_in, let's broadcast it!
50      MPI_Bcast(vector_in, COLS, MPI_INT, 0, MPI_COMM_WORLD);
51
52      // And now let's scatter the matrix row by row!
53      int* local_row = (int*) malloc(COLS * sizeof(int));
        MPI_Scatter(matrix, COLS, MPI_INT, local_row, COLS, MPI_INT, 0, MPI_COMM_WORLD);
        // XXX: Now "matrix" can be scattered as expected! But there's still a problem...

        // Now let's get the sum of products of each row by vector_in
        int sum = 0;
        for (int i = 0; i < COLS; i++) sum += local_row[i] * vector_in[i];

        // And now we gather those sums into vector_out
        MPI_Gather(&sum, 1, MPI_INT, vector_out, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if (my_rank == 0) print_matrix_and_vectors(matrix, vector_in, vector_out);

        MPI_Finalize();
        return 0;

    }
```

Use MPI_Scatter to split the matrix row by row & send each split part to a process
Each process receives one row into local_row.
COLS is the size of one row (5 integers).

Each process sends its computed sum back to process 0
Process 0 collects all sums into the array called vector_out.

2.2. State The Problem We Faced

The initial implementation made an incorrect assumption: it attempted to scatter the matrix by sending only one row per process, regardless of how many total rows the matrix contained. This caused most of the matrix rows to never be processed, producing incomplete and incorrect results in case we used number of processes less than the row size. MPI requires that each process receives equal-sized chunks during a scatter operation. Since the matrix had more rows than available processes, the correct approach was to divide the matrix into blocks of multiple rows per process, not single rows.

The program worked correctly when we used 9 processes, which matched the 9 rows in the matrix. In that case, each process received exactly one row, so the results were accurate.

However, when we used only 3 processes, the behavior became incorrect. The program still attempted to send just one row per process, meaning only three rows were processed while the remaining six rows were ignored. This produced incomplete and incorrect output.

The issue arises because MPI's scatter operation requires sending equal-sized data chunks to all processes. When the number of processes is smaller than the number of matrix rows, we must divide the matrix into blocks of multiple rows per process, not assume a "one row per process" distribution.

2.3. Output & Tests

1- Using 9 processes (equal to number of rows which is 9)

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 9 6.4.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     8
1 4 2 3 2     0     13
2 1 1 3 0           7
2 1 1 3 4           7
2 2 4 0 4           14
3 1 2 3 3           10
```

When running the program with 9 processes, the output was correct because each row of the 9 rows in the matrix was assigned to one process.

2- Using 3 processes (less than the number of rows which is 9 in our case)

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 3 6.4.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     0
1 4 2 3 2     0     0
2 1 1 3 0           0
2 1 1 3 4           0
2 2 4 0 4           0
3 1 2 3 3           0
```

When running with only 3 processes, most results became zero. This happened because the program still distributed the work as “one row per process,” so only 3 rows were processed while the remaining 6 rows were never computed. This confirmed that the implementation worked only when the number of processes equaled the number of rows and failed when fewer processes were used.

3. Solving Problem

The improved solution fixes the issue of incorrect output when the number of processes is smaller than the number of matrix rows. Instead of assigning one row to each process, the matrix is divided into equal-sized blocks of rows. Each process receives its block, performs the matrix–vector multiplication for its portion, and then sends the results back to the master process. The updated code uses rows_per_proc and elements_per_proc to properly scatter and gather matrix data, ensuring that all rows are processed correctly, even when using fewer processes.

We divide the matrix into blocks of rows, and each process receives multiple rows instead of just one.

Example:

If the matrix has 9 rows and we use 3 processes, then:

- Each process receives 3 rows (because 9/3 = 3)
- Each process multiplies its 3 rows by the vector
- Then all partial results are gathered back to the master process

This ensures:

- Every row is processed
- Any number of processes works (if ROWS is divisible by world_size)

3.1. Modified Part in Code

- **number of rows × number of columnsrows_per_proc**: This line calculates how many matrix's rows each process should work on.
- ROWS is the total number of rows in the matrix (in your code, 9).
- world_size is the total number of MPI processes running (3 for example).
- If we have 9 rows and 3 processes in the communicator so the variable rows_per_proc = 3 & that's means each process gets a block of 3 rows.
- **elements_per_proc**: calculates how many integers values each process will receive
- Each row has COLS integer elements.
- So, the total number of integers that the process receives is number of rows per process × number of columns
- If we have 3 rows per process and 5 columns in the matrix so the variable elements_per_proc = 15 & that means each process receives 15 integers

```
49      // Since they all use vector_in, let's broadcast it!
50      MPI_Bcast(vector_in, COLS, MPI_INT, 0, MPI_COMM_WORLD);
51
52      // And now let's scatter the matrix row by row!
53      int rows_per_proc = ROWS / world_size;      //  3 rows
54      int elements_per_proc = rows_per_proc * COLS; // 15 ints
55      int* local_row = (int*) malloc(elements_per_proc * sizeof(int));
56      MPI_Scatter(matrix, elements_per_proc, MPI_INT, local_row , elements_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
57      // XXX: Now "matrix" can be scattered as expected! But there's still a problem...
```

- In MPI_Scatter now, we send the elements_per_proc we explained above as the size of the buffer instead of using COLS in the old code as now we can send more than 1 row to each process.

- We need to create new array called local_results instead of just saving the local result in a variable called sum as now each process can be assigned to it more than 1 row so we need an array for each process to store in it the sum values of each row it has.
- The other modification that we now will add an outer loop for the loop we had previously which used to compute the local sums. The outer loop will be looping on the number of rows_per_proc.
- For example, if we have 3 rows per process so the local result must store 3 local sums which will be sent to vector out later.

```
59      // Now let's get the sum of products of each row by vector_in
60      int* local_results = (int*) malloc(rows_per_proc * sizeof(int));
61
62      for (int i = 0; i < rows_per_proc; i++){
63
64          local_results[i] = 0;
65
66          for (int j = 0; j < COLS; j++)
67          {
68              local_results[i] += local_row[COLS*i+j] * vector_in[j];
69          }
70      }
71
72      // And now we gather those sums into vector_out
73      MPI_Gather(local_results, rows_per_proc , MPI_INT, vector_out, rows_per_proc, MPI_INT, 0, MPI_COMM_WORLD);
74
75      if (my_rank == 0) print_matrix_and_vectors(matrix, vector_in, vector_out);
76
77      MPI_Finalize();
78      return 0;
79
80  }
```

- In MPI_Gather now, we send the local_results instead of sum so we must also change the size to be rows_per_proc instead of 1.

3.2. Output & Tests

1- Using 9 processes (equal to number of rows which is 9)

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 9 Solved.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     8
1 4 2 3 2     0     13
2 1 1 3 0           7
2 1 1 3 4           7
2 2 4 0 4           14
3 1 2 3 3           10
```

We repeat the same testcase we used in the first code and we made sure it is still working right as the previous code.

2- Using 3 processes (less than the number of rows which is 9 in our case)

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 3 Solved.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     8
1 4 2 3 2     0     13
2 1 1 3 0           7
2 1 1 3 4           7
2 2 4 0 4           14
3 1 2 3 3           10
```

Now the difference between this modified code and the previous broken one appears.

When running with only 3 processes, the result was computed right like when we used 9 processes. This happened because the matrix was divided into 3 blocks, each block containing 3 rows.

Each process performed the multiplication for its block:

- Process 0 computed rows **0–2**
- Process 1 computed rows **3–5**
- Process 2 computed rows **6–8**

The final result vector has correct values for all rows (14, 14, 5, 8, 13, 7, 7, 14, 10).

4. Difference Between Broken Code & Modified Code

4.1. Broken Code Before the Fix

Only the first 3 rows were computed (because MPI was sending only one row per process).
Rows 3–8 were all zeros.

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 3 6.4.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     0
1 4 2 3 2     0     0
2 1 1 3 0           0
2 1 1 3 4           0
2 2 4 0 4           0
3 1 2 3 3           0
```

4.2. Modified Code

Rows 3 to 8 now have correct non-zero results, proving that:

- The matrix was correctly divided into row blocks
- Every process received the right portion
- All rows were multiplied with the vector
- All results were gathered properly

```
C:\Users\Youssef Moustafa\MPI_Projects\Assignment_3>mpiexec -n 3 Solved.exe

1 2 4 0 4     0     14
4 3 3 2 4     1     14
0 0 1 2 1  *  3  =  5
1 0 2 2 1     1     8
1 4 2 3 2     0     13
2 1 1 3 0           7
2 1 1 3 4           7
2 2 4 0 4           14
3 1 2 3 3           10
```

Conclusion

In conclusion, the improved solution fixes the issue of incorrect output when the number of processes is smaller than the number of matrix rows. Instead of assigning one row to each process, the matrix is divided into equal-sized blocks of rows. Each process receives its block, performs the matrix–vector multiplication for its portion, and then sends the results back to the master process.