Chp 6 - Exercise 4 - Individual

•Implement the code for communication for chain communication.
    •This algorithm is also discussed in Chapter 8
    •The MPI_Wait should be a critical piece to ensure processes don't get information in the wrong order.
1. Comment out that part of the parallel code and see how it runs.
2. How can you ensure that the results you are getting are correct?
3. Add in a timing component to the code.
4. What sort of speed up do we see when we go from 2 to 4 processes?
5. What sort of speed up do we see when we go from 4 to 8 processes?

Grading Rubric
_____ (2 Points) Explains how commenting out the MPI_Wait affected the code
_____ (2 Points) Detail a test that would ensure the code is running correctly
_____ (2 Points) Code provided that includes the timing component
_____ (2 Points) Table that shows time for 2, 4, and 8 processors and equation and results for #4 and #5

Version 1: With MPI_Wait (Synchronized)

```
#include <algorithm>
#include <iostream>
#include <memory>
#include <random>
#include <vector>
#include "mpi.h"

void print_matrix(const float *matrix, int dim) {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      std::cout << matrix[i * dim + j] << ' ';
    }
    std::cout << '\n';
  }
}

int main(int argc, char *argv[]) {
  // Initialize MPI
  MPI_Init(&argc, &argv);

  // Get the total number of tasks
```

```
  int num_tasks;
  MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

  // Calculate the number of rows mapped to each process
  const int dim = 1 << 12;
  const int n_rows = dim / num_tasks;

  // Get the task ID
  int task_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &task_id);
  const int start_row = task_id * n_rows;
  const int end_row = start_row + n_rows;

  // Matrix - Only initialized in rank 0
  std::unique_ptr<float[]> matrix;

  // Each process will store a chunk of the matrix
  auto m_chunk = std::make_unique<float[]>(dim * n_rows);

  // Each process will receive a pivot row each iteration
  auto pivot_row = std::make_unique<float[]>(dim);

  // Only rank 0 create/initializes the matrix
  if (task_id == 0) {
    std::mt19937 mt(123);
    std::uniform_real_distribution dist(1.0f, 2.0f);
    matrix = std::make_unique<float[]>(dim * dim);
    std::generate(matrix.get(), matrix.get() + dim * dim, [&]
{ return dist(mt); });
  }

  // Scatter parts of the matrix to each process
  MPI_Scatter(matrix.get(), dim * n_rows, MPI_FLOAT,
m_chunk.get(), dim * n_rows, MPI_FLOAT, 0, MPI_COMM_WORLD);

  // Timing component
  double start_time = MPI_Wtime();

  std::vector<MPI_Request> requests(num_tasks);

  // Gaussian elimination loop
  for (int row = 0; row < end_row; row++) {
    auto mapped_rank = row / n_rows;
```

```
    if (task_id == mapped_rank) {
      auto local_row = row % n_rows;
      auto pivot = m_chunk[local_row * dim + row];

      for (int col = row; col < dim; col++) {
        m_chunk[local_row * dim + col] /= pivot;
      }

      for (int i = mapped_rank + 1; i < num_tasks; i++) {
        MPI_Isend(m_chunk.get() + dim * local_row, dim,
MPI_FLOAT, i, 0, MPI_COMM_WORLD, &requests[i]);
      }

      for (int elim_row = local_row + 1; elim_row < n_rows;
elim_row++) {
        auto scale = m_chunk[elim_row * dim + row];
        for (int col = row; col < dim; col++) {
          m_chunk[elim_row * dim + col] -= m_chunk[local_row *
dim + col] * scale;
        }
      }

      // Use MPI_Wait to ensure synchronization
      for (int i = mapped_rank + 1; i < num_tasks; i++) {
        MPI_Wait(&requests[i], MPI_STATUS_IGNORE);
      }
    } else {
      MPI_Recv(pivot_row.get(), dim, MPI_FLOAT, mapped_rank, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      for (int elim_row = 0; elim_row < n_rows; elim_row++) {
        auto scale = m_chunk[elim_row * dim + row];
        for (int col = row; col < dim; col++) {
          m_chunk[elim_row * dim + col] -= pivot_row[col] *
scale;
        }
      }
    }
  }

  // Gather the final results into rank 0
  MPI_Gather(m_chunk.get(), n_rows * dim, MPI_FLOAT,
matrix.get(), n_rows * dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

```
  double end_time = MPI_Wtime();
  double elapsed_time = end_time - start_time;

  if (task_id == 0) {
    std::cout << "Time taken with " << num_tasks << "
processes: " << elapsed_time << " seconds\n";
  }

  MPI_Finalize();
  return 0;
}
```

Version 2: Without MPI_Wait (Unsynchronized)

```
#include <algorithm>
#include <iostream>
#include <memory>
#include <random>
#include <vector>
#include "mpi.h"

void print_matrix(const float *matrix, int dim) {
  for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
      std::cout << matrix[i * dim + j] << ' ';
    }
    std::cout << '\n';
  }
}

int main(int argc, char *argv[]) {
  // Initialize MPI
  MPI_Init(&argc, &argv);

  // Get the total number of tasks
  int num_tasks;
  MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);

  // Calculate the number of rows mapped to each process
  const int dim = 1 << 12;
  const int n_rows = dim / num_tasks;

  // Get the task ID
```

```
  int task_id;
  MPI_Comm_rank(MPI_COMM_WORLD, &task_id);
  const int start_row = task_id * n_rows;
  const int end_row = start_row + n_rows;

  // Matrix - Only initialized in rank 0
  std::unique_ptr<float[]> matrix;

  // Each process will store a chunk of the matrix
  auto m_chunk = std::make_unique<float[]>(dim * n_rows);

  // Each process will receive a pivot row each iteration
  auto pivot_row = std::make_unique<float[]>(dim);

  // Only rank 0 create/initializes the matrix
  if (task_id == 0) {
    std::mt19937 mt(123);
    std::uniform_real_distribution dist(1.0f, 2.0f);
    matrix = std::make_unique<float[]>(dim * dim);
    std::generate(matrix.get(), matrix.get() + dim * dim, [&]
{ return dist(mt); });
  }

  // Scatter parts of the matrix to each process
  MPI_Scatter(matrix.get(), dim * n_rows, MPI_FLOAT,
m_chunk.get(), dim * n_rows, MPI_FLOAT, 0, MPI_COMM_WORLD);

  // Timing component
  double start_time = MPI_Wtime();

  std::vector<MPI_Request> requests(num_tasks);

  // Gaussian elimination loop
  for (int row = 0; row < end_row; row++) {
    auto mapped_rank = row / n_rows;

    if (task_id == mapped_rank) {
      auto local_row = row % n_rows;
      auto pivot = m_chunk[local_row * dim + row];

      for (int col = row; col < dim; col++) {
        m_chunk[local_row * dim + col] /= pivot;
      }
```

```
      for (int i = mapped_rank + 1; i < num_tasks; i++) {
        MPI_Isend(m_chunk.get() + dim * local_row, dim,
MPI_FLOAT, i, 0, MPI_COMM_WORLD, &requests[i]);
      }

      for (int elim_row = local_row + 1; elim_row < n_rows;
elim_row++) {
        auto scale = m_chunk[elim_row * dim + row];
        for (int col = row; col < dim; col++) {
          m_chunk[elim_row * dim + col] -= m_chunk[local_row *
dim + col] * scale;
        }
      }

      // Commenting out MPI_Wait
      /*
      for (int i = mapped_rank + 1; i < num_tasks; i++) {
        MPI_Wait(&requests[i], MPI_STATUS_IGNORE);
      }
      */
    } else {
      MPI_Recv(pivot_row.get(), dim, MPI_FLOAT, mapped_rank, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      for (int elim_row = 0; elim_row < n_rows; elim_row++) {
        auto scale = m_chunk[elim_row * dim + row];
        for (int col = row; col < dim; col++) {
          m_chunk[elim_row * dim + col] -= pivot_row[col] *
scale;
        }
      }
    }
  }

  // Gather the final results into rank 0
  MPI_Gather(m_chunk.get(), n_rows * dim, MPI_FLOAT,
matrix.get(), n_rows * dim, MPI_FLOAT, 0, MPI_COMM_WORLD);

  double end_time = MPI_Wtime();
  double elapsed_time = end_time - start_time;

  if (task_id == 0) {
    std::cout << "Time taken with " << num_tasks << "
```

```
processes: " << elapsed_time << " seconds\n";
  }

  MPI_Finalize();
  return 0;
}
```

Run time at 2:

```
sa7233@mct263s02:~/fall2024/HPC/ch6$ mpirun -n 2 ./
with_MPI_wait
Time taken with 2 processes: 436.685 seconds
```

Run time at 4:

```
sa7233@clipper:~/fall2024/HPC/ch6$ mpirun -n 4 ./with_MPI_wait
Time taken with 4 processes: 148.079 seconds
```

Run time at 6:

```
sa7233@mct263s02:~/fall2024/HPC/ch6$ mpirun -n 6 ./
with_MPI_wait
Time taken with 6 processes: 164.574 seconds ( this test was on
the machine in mct263)
```

Run time at 8:

```
sa7233@clipper:~/fall2024/HPC/ch6$ mpirun -n 8 ./with_MPI_wait
Time taken with 8 processes: 81.053 seconds
```

Run time at 20:

```
sa7233@clipper:~/fall2024/HPC/ch6$ mpirun -n 20 ./with_MPI_wait
Time taken with 20 processes: 31.8416 seconds
```

** just for fun to see if can run in single digits

Table based on the runtime data for different numbers of processes:

Processes	Run Time (seconds)
2	436.6850
4	148.0790
6	164.5740
8	81.0530
20	31.8416

In this exercise, two versions of the code were tested:

Version 1 (Synchronized using MPI_Wait): This version ensures that processes wait for communication to complete before proceeding to the next iteration, guaranteeing that the data is received in the correct order.

Version 2 (Unsynchronized, MPI_Wait commented out): In this version, the MPI_Wait calls were removed/commented out, allowing processes to move forward without waiting for communication to finish. This tests how the program behaves when communications aren't properly synchronized.

Key Observations:

MPI_Wait's Role: When MPI_Wait is commented out, the processes may receive data in the wrong order, leading to incorrect results. MPI_Wait acts like a checkpoint, ensuring that each process receives the correct pivot row before proceeding with Gaussian elimination. Without it, processes proceed asynchronously, causing potential issues where one process is ahead of others and operates on incorrect or incomplete data.

To illustrate, think of two baskets of balls (one red, one blue) being rolled down a set of stairs. Without MPI_Wait, there is no guarantee that the red and blue balls will bounce the same way, and they may land at the bottom out of order. MPI_Wait ensures that the balls land exactly in sync.

Timing and Speedup Analysis:

- With 2 processes, the runtime was **436.685 seconds**.
- With 4 processes, the runtime decreased significantly to **148.079 seconds**, showing a **3x speedup**.
- With 8 processes, the runtime reduced further to **81.053 seconds**, representing roughly **half the time** compared to 4 processes.

This demonstrates that increasing the number of processes from 2 to 4 yields a clear improvement, with execution time reducing by a factor of 3. However, the speedup from 4 to 8 processes shows diminishing returns, as the runtime only decreases by half, indicating that as more processors are added, the overhead of managing communication increases and limits the performance gains. Also, I would like to point out that the size was 2^12 which is 4096 so bigger size might benefits from more processors compared to this size.