CS328 · Distributed & Cloud Computing

Assignment 1

Yaozhong Kang / 12110225

October 15, 2023

1 Design

We're required to design a matrix multiplication method using **Open MPI** to meet the following requirements:

- Use two matrices, A and B, both of size 500×500 with elements of type double.
- The **master process** should:
  - Control data distribution and gathering.
  - Distribute computation to worker processes and itself.
  - Compare results with a brute force approach to verify correctness.
- Design a mechanism for **non-even matrix splitting**.
- Run the program with 1, 2, 4, 8, 16, and 32 MPI processes on:
  - One virtual machine (VM).
  - A 2-VM cluster.
- Analyze and explain performance based on plotted figures.
- **Bonus**: Suggest optimizations for improved computation efficiency.

1.1 mpi_matrix_solved.c

1. **Initialization:**
   Initialize the matrices by reading them from files "matrixA.txt" and "matrixB.txt". We're not using dynamic memory allocating.
2. **Data distributing and Gathering**:
   We should consider the **non-even matrix splitting** before we do the distributing. From the document, we can find methods dedicated for evenly distributing and gathering data.

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

   However, this two kinds of method can only split the resource data evenly. In case of doing non-even splitting, I considered that using for loop and send-receive process would be more simple and won't be more consumptive.
   If there's only one task (process), it runs the matrix multiplication sequentially.
   The code uses the offset value and the remainder value to store position of the pointer where we should pass the data from.
   Considering the matrix on the left:

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}$$

The threads will by turn taking care of each row. For example, if we have three threads, the first row will be pass to the first thread, second row to the second...and the fourth row back to the first thread etc.

3. **Local Multiplication**:
   - After sending the data, the master waits for results from the worker processes.
   - The master process initializes the matrices, while the slave processes will be used to compute partial results.
   - Once results are received from all workers, it checks the MPI-based multiplication result against a brute-force method for verification.
   - The approach used here for matrix multiplication is not the most efficient. More sophisticated algorithms and data distribution methods can further optimize performance.

1.2 mpi_multiply_fox.c

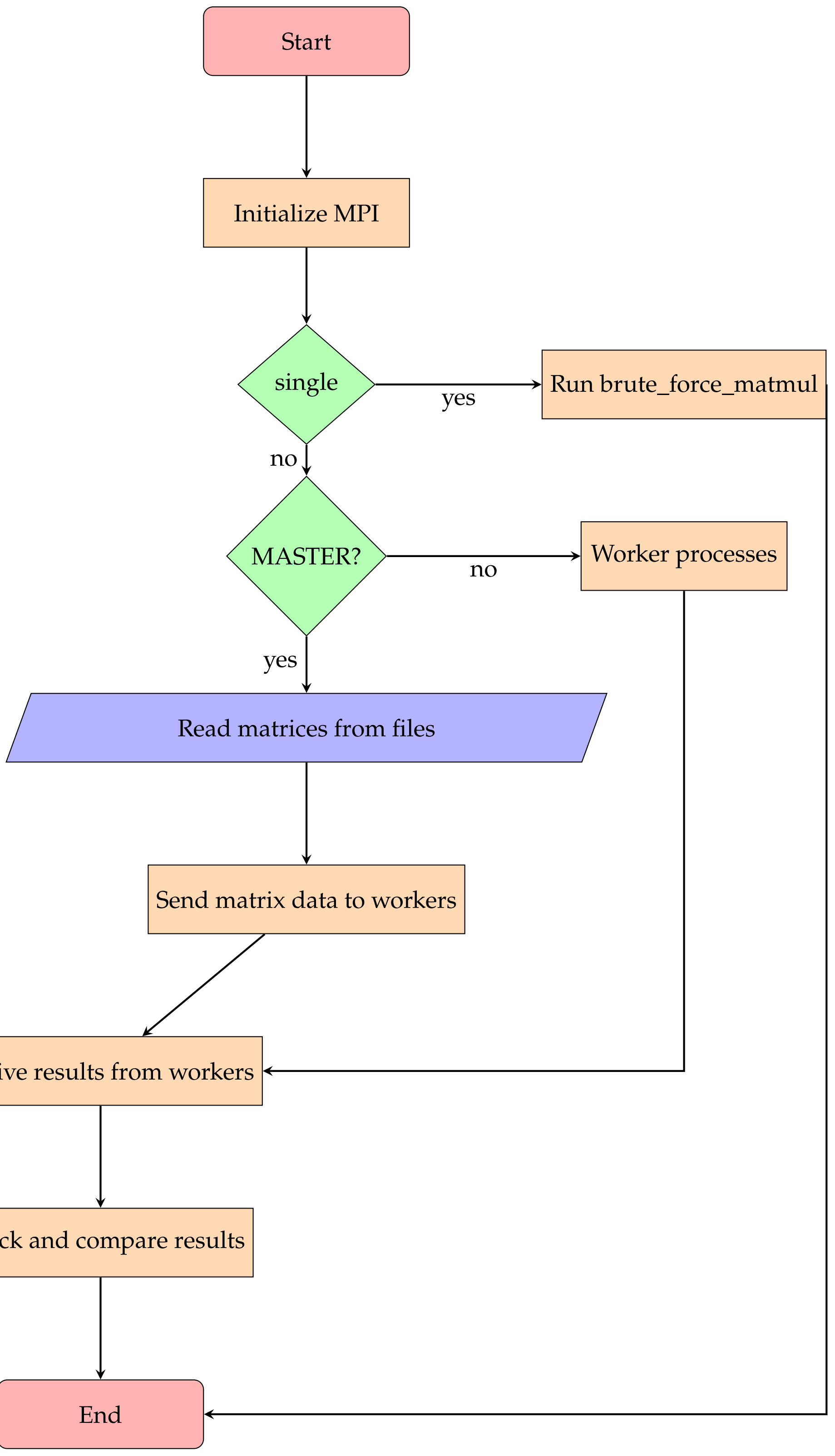
This piece of code is **NOT TOTALLY IMPLEMENTED BY ME**. Most part of the code is from example from the lecture code of *COMP 705: Advanced Parallel Computing*. It's an optimization for improved computation efficiency using block-wise distributing.

1. **Matrix and Grid Definitions:**
   Structures GRID_INFO_T and LOCAL_MATRIX_T are used to define the grid and local matrices properties, respectively. The grid information includes the total number of processes, communicators for rows and columns, and the order of the grid. The matrix structure holds information about the order of the matrix and its entries.

2. **Data Distribution:**
   Unlike a basic scatter and gather, the implementation uses Fox's algorithm, which divides matrices into smaller square blocks and efficiently multiplies them in parallel. This involves a series of broadcast and shift operations over stages to ensure each process has the correct data to compute with

```
void Fox(int n, GRID_INFO_T *grid, LOCAL_MATRIX_T *local_A,
        LOCAL_MATRIX_T *local_B, LOCAL_MATRIX_T *local_C)
```

3. **Matrix Multiplication Process:**
   In each stage of the Fox algorithm, a portion of matrix A is broadcasted within its row. Then, each process multiplies its current sub-matrices and accumulates the result. Sub-matrix B undergoes a circular shift for the next stage.
   Reference: *G. Fox, et. al., "Matrix algorithms on a hypercube I: Matrix Multiplication"*



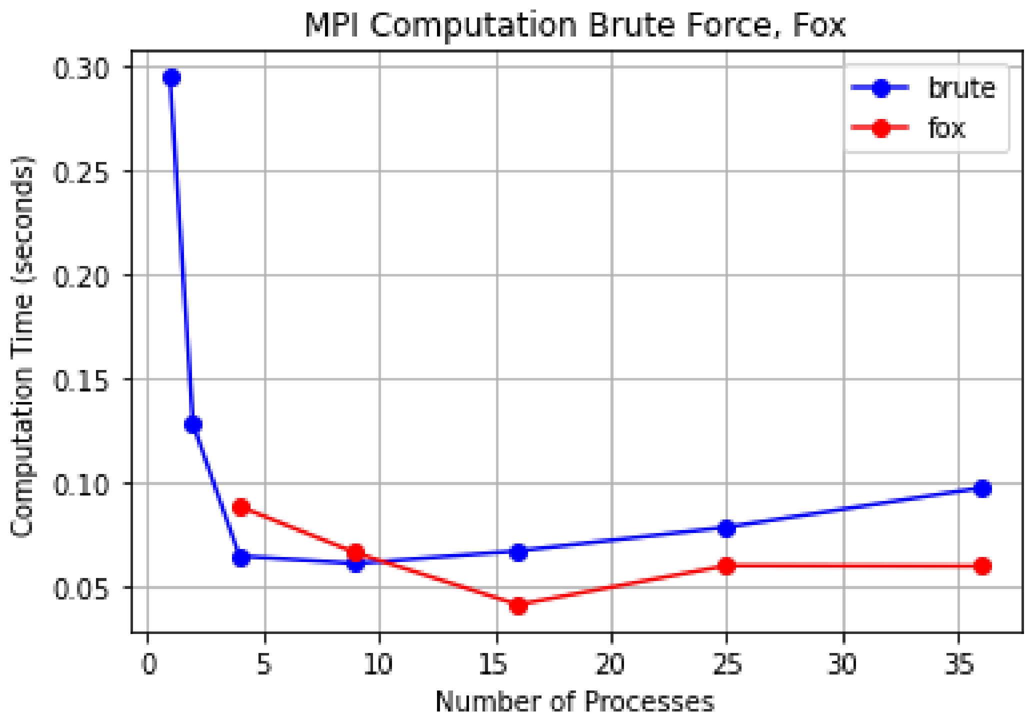
2 Running Result

You can explore around the code in *My Repo*. Also, you can generate the graph using test.py in the code. In the first test, we're using a 8 core intel laptop which claims to have 16 threads.

```
Architecture:          x86_64
CPU op-mode(s):          32-bit, 64-bit
```

```
Address sizes:        39 bits physical, 48 bits virtual
Byte Order:           Little Endian
CPU(s):               16
On-line CPU(s) list:  0-15
Vendor ID:            GenuineIntel
Model name:           13th Gen Intel(R) Core(TM) i5-13500H
```

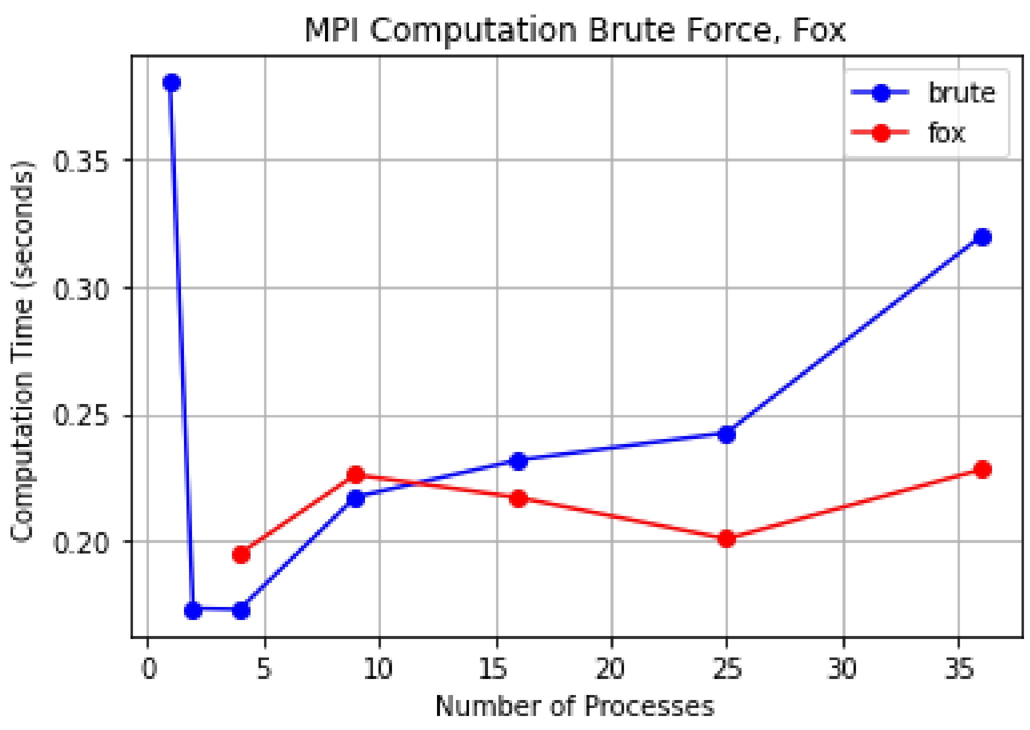
The illustration of the Computations Time VS Number of Processes:



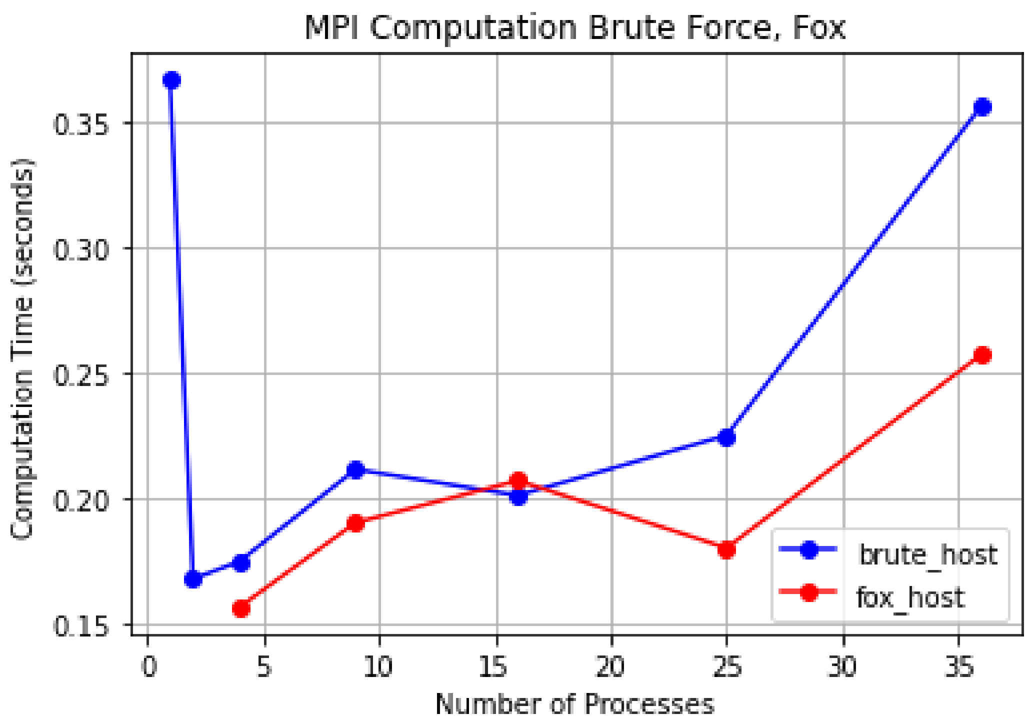
The Computation Time goes down as the process increases. Focusing on the blue curve we find that between 9 and 16 processes the program uses least time. When the process goes beyond 16, the slots are fake and cause more communication consumption.
The illustration of the Computations Time VS Number of Processes and using 2 VMs
We're using two 8-core virtual machines on an Intel desktop.

```
Architecture:         x86_64
CPU op-mode(s):       32-bit, 64-bit
Address sizes:        45 bits physical, 48 bits virtual
Byte Order:           Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Vendor ID:            GenuineIntel
Model name:           13th Gen Intel(R) Core(TM) i7-13700KF
```

The code working on one VM:



The code working cooperatively on two VMs multiplying the same matrix as previous machine:



We can see that when the amount of slots is lower or equal than 25.Two VMs works better than single machine. However, they cannot reach the performance to twice as fast as the single VM.

3 Problems

- Dynamic Memory Allocating
  Memory allocation is quite tricky in open-mpi. Most of the problems are resulted from serialization. Some data would be lost if the data being transferred is too large using MPI_send and MPI_Receive method.
- Optimization Requires a Good Matrix The "fox" method optimizes matrix multiplication when the matrix grows bigger. However, the size of the matrix must be divisible by the square root of the amount of processes.
  Thus, in the python script, we tested on 1, 4, 9, 16, 25 and 36 threads. And we set the size of matrix to 480.
- Platform matters in open-mpi
  mpirun would ran into problems using MacOS. Especially when occupying more threads.