Comp 4510 – Assignment 2

Due: October 19, 2022(11:59pm)

Instructions:

This assignment should be completed on the Mercury machines using MPI. The *MPI Programming Environment* document on the course website describes how to log into these machines and use them – please read it before you begin.

You will be expected to follow the *Assignment Guidelines*, which are available in the assignments folder on the course website. This document describes how to organize and hand in your code. Please review it before you begin. Note that a failure to follow these standards will result in a loss of marks. You may place all the written part of the questions in one single document. Ensure you properly label the questions in the document. Always make sure your program works for any input and ***optimize*** your code. Submit all programs that you have written or made modifications to.

This assignment helps you understand the single program multiple data approach, static and dynamic (manager-worker). It also helps to understand MPI blocking and non-blocking routines.

Total: [40] marks.

I. Programming Questions:

1. [10] Consider the program we discussed in class from LAM-MPI manual (page 12). The example was to illustrate dynamic load balancing using an on-demand approach. We will now look at converting this program into a static load balancing approach, in which all processes participate in the computation. There is no need for tags in the modified program. You may use any other variables (if necessary) as per the manual. The algorithm is as follows:
    i. Given `NUM_WORK_REQs` (say `N`) and `q` processes. Assume, `N,` the number of exam papers, and `q`, the number of markers.
    ii. We could open a file and ask each process to read in the marks for their individual papers. However, we will make it simple and will let process 0 randomly fill the array of size `N` with values between 0 and 100% (these could be floating point numbers also), the mark for each of the `N` students.
    iii. Process 0 performs distributes the `N/q` exam papers using a block data decomposition.
    iv. Each process (including process 0) computes the sum of the marks for their papers (call the variable `sub-total`).
    v. Process 0
        i. collects the sub-`total` from each process and computes the sum of the overall total (call it `total`) using collective communication functions.
        ii. calculates the average (call it `average`) from `total.`
        iii. distributes the average to all processes using collective communication functions.
    vi. Each process prints the following message "I am process `xxx` with `yyy` number of exam papers. The average is `zzz` ."

    (NOTE: `N` need not be divisible by `q`. Your program should work for all values of `N` and `q`. If you are using `MPI_Scatter` or `MPI_Gather,` there are limitations that each process must send or receive the same number of data items. When this is not the case, we must use `MPI_Scatterv`

or `MPI_Gatherv`. You may also want to check out `MPI_Reduce_scatter`. Look at the manual pages for more information.)

2. [12] In this question, you will use MPI to implement matrix-vector multiplication in parallel. Let `A` be a matrix of size *n* x *n*, and let `b` be a vector of size *n* x 1. Then we can multiply `A` · `b` as shown below, the result of which is an *n* x 1 vector `c`.

$$\begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{n0} & a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{00}b_0 + a_{01}b_1 + \cdots + a_{0n}b_n \\ a_{10}b_0 + a_{11}b_1 + \dots + a_{1n}b_n \\ \dots \\ a_{n0}b_0 + a_{n1}b_1 + \dots + a_{nn}b_n \end{bmatrix}$$

(A) · (b) = (c)

Implement one program for each of the questions (a) and (b) below. For both questions, process 0 should collect the results and print out the `c` vector in the manner described in the "Implementation Details" section below. Your program should be capable of running with any *n* and *q*, where $q < n$, [*n* is the number of rows in `A` and *q* is the number of processes]. You may assume *n* is evenly divisible by *q*.

a) [5] On demand strategy: You may declare process 0 as the manager process. Each process receives one row of `A`. Vector `b` is available to all processes. Each process should compute one element of `c`.

b) [5] Assume the size of `A` and `b` are extremely large. In this case, we follow a block data distribution for both `A` and `b` by partitioning `A` and `b` row-wise. We usually open a file for each process to read in their respective values of `A` and `b`. However, in this problem, we will assume process 0 randomly fills in `A` and `b` as per implementation details below. Process 0 distributes *n* /*q* rows of A and *n* / *q* elements of `b` to each process using a block data decomposition. Each process now operates on *n* /*q* rows of `A` and *n* / *q* elements of `b` and computes `c` for *its n/q* elements. [Note: you will need to figure out how to communicate the required data between the processes after the initial partitioning has occurred.]

c) [2] What are the challenges in the parallel program implementation if we followed a cyclic data decomposition strategy? That is, what changes would you need to make to your program in b) using a cyclic data decomposition strategy?

Implementation Details:

- No special data structures are required – you can store the matrix and vector values in regular arrays. You may find it easier to work with flat (1D) arrays instead of 2D arrays, since the latter can be awkward in C, but it is up to you.
- Generate `A` and `b` using the standard C `rand()` function. **Mod the values you generate by 10 before you insert them into the array (this will keep them small). You should call `srand(1)` first to seed the generator with the constant 1.** This will allow the marker to check your solution.
- When your program completes, the following information should be printed to standard output:
    - The values of *n* and *q*.

- The execution time, **excluding** any time required to initialize `A` and `b`. The vec_sum MPI example on the course website demonstrates how to use MPI timing functions to measure execution time.
- The `c` vector (one element per line).

3. [3] Consider the serial sorting algorithm bubble sort. Write the sequential code (you may retrieve it from any textbook) and indicate clearly whether there are opportunities for parallelism in this serial algorithm.

4. [5] Write a parallel program using **non-blocking MPI functions** only. Process `i` sends its rank to process `(i+1)%q`. Each process prints the following output, "I, process `xxx`, received a rank from my neighbouring process `yyy`".

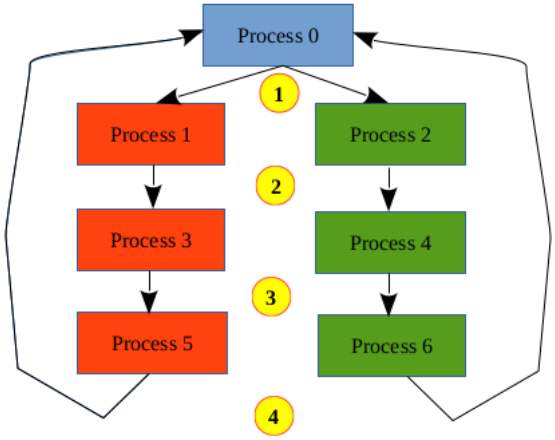
5. [10] In this question, you will use message passing functions to simulate a race between two teams of processes. To do this, process 0 will be considered the "race official", and we'll divide the remaining processes into two teams by parity (even-numbered processes on one team, and odd-numbered processes on the other). [NOTE: If you want to learn advanced MPI functions, you may use `MPI_comm_group()` and other associated library functions to implement this by studying the MPI manual.]

   Therefore, to ensure that both teams have an equal number of processes, the total number of processes (`q`) will be an odd number. For example, if `q` = 7, the processes are split up as follows:

   Odd Team: 1, 3, 5
   Even Team: 2, 4, 6
   Race Official: 0

   a. [6] Simulate the race by following this approach:

- Process 0 will start by generating an array of size `n` integers. Start by assuming that `n` = 10 (we will try varying it later). These integers do not have to be random – we just want some bytes to pass around.
- Our communication structure will look like this:



- Process 0 will start a timer. Then it will send the array to processes 1 and 2 (the starters for the two teams). This send should happen *asynchronously*.

- Process 0 should then wait to receive the data back from either team. When it receives data, it should identify which team it is from, grab the current value of the timer and calculate the total completion time for the team. It should then wait to do the same for the other team.
- Team member processes should wait to receive the data and then pass it on to their next team member (in increasing order by rank, as shown in the diagram above). Note that the last processes on each team must send the data back to process 0.

After both teams complete, process 0 should print the following output (your results may be different):

```
q = 7
n = 10
Even team time:1.234sec
Odd team time: 4.321sec
Even team wins!
```

b. [1] Run your program with the settings shown in the example above (`q` = 7, n = 10) and record the winning time.
c. [2] Run your program again with each of the following settings, recording the winning times:
   i. Try doubling `q` (and adding one for the race official) and leaving `n` the same. Use the parameters (`q` = 15, n = 10).
   ii. Now, try doubling `n` (and adding one to be consistent) and leaving `q` the same. Use the parameters (`q` = 7, `n` = 21).
d. [1] Look at the winning times you recorded in part (b). Which causes the winning time to increase more: doubling `q` (i) or doubling `n` (ii)? Comment on why you think this might be the case.