Assignment 5 Report - Distributed Computing with Message Passing

Dean Tsankov, Kacper Lisik

March 2, 2025

Introduction

In this assignment, we learn how to program and use message passing in a distributed application, as well as to program, build, and execute distributed programs in C with the MPI library.

Problem 1 - Distributed Pairing 1 (a client-server application)

Task description

```
Assume that a class has n students numbered 1 to n and one teacher. The
↪ teacher wants to assign the students to groups of two for a project.
↪ The teacher does so by having every student submit a request for a
↪ partner. The teacher takes the first two requests, forms a group from
↪ those students, and lets each student know their partner. The teacher
↪ then takes the following two requests, forms a second group, and so
↪ on. If n is odd, the last student "partners" with themself.
```

Key points of solution

The solution consists of creating two types of processes and allowing them to communicate asynchronously using massage passing. The teaher process simply iterates through the number of students and waits to receive a request from any of them, not particularly in order of rank by utilising the `MPI_ANY_SOURCE` wildcard in its source parameter field. It then keeps track of how many students have been unpaired and if that number reaches 2 we now that we can pair the last two unpaired students by sending them a message including the respective other student's rank as on argument. Later there is one more check wether there was a single student left without a partner

(when the total number of students is odd) we pair that one with itself. Finaly we send a massage to all students that the teacher is done and that signals each of them to share who they were paired with. This is done since in the problem instructions we are told to do the final printout after the program is done not while the allocation is happening, which is also an option.

Here is the code for the teacher function:

```
(...)
void teacher(int stNum)
{
    int i;
    int students[stNum];
    MPI_Status status;
    int unpaired = 0; // increment when a message is recieved and if
    ↪  it gets to two then pair the current student with the
    ↪  previous one

    for (i = 0; i < stNum; i++)
    {
        //receive message from any source since we do not know which
        ↪  student will request first
        MPI_Recv(&students[i], 1, MPI_INT, MPI_ANY_SOURCE, 0,
        ↪  MPI_COMM_WORLD, &status);
        printf("Received request from %d\n", students[i]);
        unpaired++;
        printf("Curently unpaired %d\n", unpaired);
        if (unpaired == 2)
        {
            MPI_Send(&students[i], 1, MPI_INT, students[i - 1], 0,
            ↪  MPI_COMM_WORLD);
            MPI_Send(&students[i - 1], 1, MPI_INT, students[i], 0,
            ↪  MPI_COMM_WORLD);
            unpaired = 0;
        }
    }
    //if any student was left unpaired (odd num of students) => pair
    ↪  by itself
    if(unpaired == 1){
        MPI_Send(&students[stNum-1], 1, MPI_INT, students[stNum - 1],
        ↪  0, MPI_COMM_WORLD);
    }

    for (i = 0; i < stNum;i++){
        int done = 1;
```

```
            MPI_Send(&done, 1, MPI_INT, students[i], 0, MPI_COMM_WORLD);
        }
    }
    (...)
```

Next is the student process which is fairly simple since it essentially just sends a massage with its own rank number or id and then waits to receive a partner from the teacher process with the dedicated rank 0. Lastly in order for the printout to happen after the teacher has finished assigning pairs we wait to recieve another massage.

```
    (...)
    void student(int id)
    {
        int pairStId;
        MPI_Status status;

        //ask teacher process rank 0 for a pairing
        MPI_Send(&id, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

        //receive pairing from teacher
        MPI_Recv(&pairStId, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        //wait for teacher to pair evryone before printing
        int teacherDone = 0;
        MPI_Recv(&teacherDone, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
        ↪  &status);

        if(teacherDone){
            printf("The student %d was paired with the student %d\n", id,
            ↪  pairStId);
        }
    }
    (...)
```

Ultimately the main method has the following simple form:

```
    (...)
    int main(argc, argv)
    int argc;
    char **argv;
    {
        int rank, size;
```

```
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        //the 0 rank process will be the teacher, anything else is a
        ↪  student
        if(!rank){
            //exclude teacher from the student pool
            teacher(size-1);
        }else{
            student(rank);
        }

        MPI_Finalize();
        return 0;
    }
    (...)
```

Problem 6 - Exchanging Values

Task description

```
Consider the three program outlines (algorithms) in Lecture 15, slides
↪  28-34, for exchanging values by interactive peers. Implement each
↪  algorithm in C using the MPI library or in Java using the socket API
↪  or Java RMI. You may use collective communication (or multicast in
↪  Java) where appropriate. Each program should execute a sequence of R
↪  rounds of exchanges. Your program should print a trace of the
↪  interesting events as they happen, but do not make the trace too
↪  verbose. Estimate and compare the performance of your programs. Build
↪  a plot that shows the total execution time of each program as a
↪  function of the number of processes (2, 4, 6, 8) and the number of
↪  rounds (1, 2, 3). When performing evaluation experiments, do not
↪  print a trace of events.
```

Key points of solution

The solution to the problem follows the directions explained in the lecture slides quite closely. As such the implementation consists of the three different approaches:

- Centralized.
- Circular.
- Symmetric.

We will ommit pasting their code here since each of their respective code programs can be found on the assignment's github repository.

After executing the code we can visualize the outputs in the graph bellow, which follows the general tendencies we would expect even while the data might seem a bit sporadic.