Assignment 5- MPI

Emanuel Paraschiv & Mohammad Asfour

February 2025

Introduction

The scope of this assignment is to program distributed systems using message passing. We use C with the MPI library to program these distributed applications

Distributed Pairing 1

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 themselves.

In order to attain it, each process will have its own copy of the program. TO identify itself, we consider the server process(teacher) to have rank 0, master.

```
if (rank == 0) {
    teacher(n);
} else {
    students(rank);
}
```

The teacher will wait until a message is received from one of the students, and will store each requester's rank in an array.

```
for (int i = 0; i < n; i++) {
    MPI_Recv(&student_rank, 1, MPI_INT, i + 1, 0, MPI_COMM_WORLD, &status);
    arr[i] = student_rank;
}
```

After all messages are received, it will group every 2 students by sending to each of them the rank of the other.

Client processes (students) can be seen as wrapping this process: sending a request and waiting for a response. When the server provides it, the process also prints information about the given student and its assigned partner.

```
MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

// receive partner from the teacher
MPI_Recv(&partner, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

printf("Student %d is paired with Student %d\n", rank, partner);
```

Distributed Pairing 2

As in the previous problem, we want to assign n students numbered 1 to n to groups of two for a project. However, the teacher does not want to be involved in forming the groups. Instead, he asks the students to interact with each other to pair up into groups. In particular, the students must devise a single algorithm every student executes. Develop a distributed pairing algorithm for the students. An ideal solution will use only n messages. The algorithm should use random numbers so that the outcome is non-deterministic. The teacher starts the algorithm by picking one student randomly and sending a message to that student saying, in effect, "your turn to pick a partner."

As before, there is a server and multiple clients. However the server only initiates the program, then lets the clients continue it on their own.

The sever will create a shared buffer which will contain all the students. All elements(excepts for the server itself) are initialized to 0, meaning that they are free. Then the teacher will randomly pick a student to start, by sending them the availability array:

```
while (students[first_student] != 0) {
    first_student = (first_student + 1) % size;
}
MPI_Send(students, size, MPI_INT, first_student, 0, MPI_COMM_WORLD);
```

As far as student processes are concerned, they are blocked until they received their own version of student array (passed to one process at a time).

When they receive the copy, they will check if they are already paired. If not, then they will mark themselves as 'taken' in the array and will try to find the next free process.

```
if (students[rank] == 0) {// Does not have a partner
    students[rank] = rank;

    int next_exists = 0;
    for (int i = 1; i < size; i++) {
        if (students[i] == 0) {
```

```
            next_exists = 1;
            break;
        }
    }
```

If there is no ne left, that student will work alone. If however it finds some free partners, they will choose one randomly and mark both of them as paired.

After that, it will try to find the next idle student and send them the updated array, letting the process continue until everyone is grouped.

Dining philosophers

The dining philosophers problem is a common problem in concurrent application/distributed systems. It describes a problem where you have 5 philosophers (usually n) that have to eat spaghetti with 5 forks (usually n) on the table. Each philosopher needs 2 forks at the same time to eat, so synchronization must be done to avoid deadlock. But because of how message passing works, we do not need any synchronization, since the server takes one request at a time.

First, philosophers will think then send a request for acquiring forks to the server

```
message = REQUEST;
MPI_Send(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

The server keeps a record of available forks, and when it gets the request it check if the forks on the left and right of this philosopher is available. If yes it "gives them the forks":

```
if (forks[left] && forks[right]) {
forks[left] = forks[right] = 0; //lock corresponding forks
response_type = APPROVED;
}
```

The philosopher will receive the response, and eat if APPROVED, or retry after thinking if DENIED.

Each philosopher eats n times (in this case 3) before exiting the while loop and terminating the program (it sends a message to the server that it is done).