National University of Singapore
School of Computing

CS3210 Assignment 2

# Playing multi-process football

through Message Passing Interface (MPI)

*Syed Abdullah*

November 20, 2017

# Contents

# Chapter 1

# Preamble

In this report, we will be investigating MPI (implementation: OpenMPI) point-to-point and collective communication through the use of a football game.

## 1.1 System Specifications

The machines that this program will be benchmarked on is the three system set up in the Parallel and Distributed Computing Lab in SoC. The CPU specifications of the three system set up are as follows:

- i5 Machine – Intel Core i5-2400 (2.5GHz, 4 cores)
- i7 Machine – Intel Core i7-2600 (3.4GHz, (4×2) cores with HyperThreading)
- Jetson – ARM Cortex-A15 CPU (4 cores)
- NSCC normal queue – 2 × Intel E5-2690 v3 (2.60GHz, 12 cores)

# Chapter 2

# Training – Process-to-Process Communication

In the first half of this report, we will look into point-to-point communication using OpenMPI. Do note that all relevant source code are located in either: `common` or `training` folders. The header files and data structure definitions can be found in the `headers` folder.

## 2.1 Overview

In the training session, we can split each round into three phases: ball 'knowledge' phase, player state update phase and print phase. With the exception of the print phase, which is only executed by the field process, all processes have a part to play in the preceding two phases.

At the start of the program, all players are distributed across the field in a deterministic fashion, according to the player's process rank. This would allow us to cut down on the communication needed to set the initial position of the players.

## 2.2 The Field

```
1 instantiate players 'state'
2 while (there exists rounds) {
3     MPI_Send ball coordinates to player processes
4     MPI_Recv two coords from each player: [(new position), (new ball position)]
5     Determine the player who "gets" the ball (through random)
6     Update relevant statistics for player
7     Print out round statistics
8 }
```

Figure 2.1: Sketch of implementation for field process (`field_training.c`)

The field process is in charge of 'remembering' the current state of the game (i.e. where the players are at and relevant statistics on players). This is done through an array of `player` struct (defined in `datastructs.h`), called `players`. The struct basically stores the player details that are printed out at the end of the round (for instance: previous coordinates, current coordinates and so on).

### 2.2.1 Information about the Ball

The player at index 0 is the ball and the only member of `player` struct that is used for this 'player' is the position. While using a separate variable for the 'ball' location would be ideal, this 'padding' of the `players` array would allow us to obtain/set a player's state using the player's MPI rank, rather than having to remember

to 'offset' the value by one. It also allows us to reduce the use of global variables and yet, pass fewer variables to other functions.

### 2.2.2 Determining the 'winner' of the ball

According to the specifications, the 'winner' of the ball is chosen randomly and that there's no concrete requirement on the type of randomness that needs to be achieved. As such, this field process keeps track of a tentative winner and would allow subsequent players to challenge the ball through checking if the random number being generated is an even number. This allows us to keep track of the current winner, rather than an array of players which are on the ball.

## 2.3 The Player

```
1  set player's initial coordinates
2  while (there exists rounds) {
3      MPI_Recv ball coordinates from field process
4      Calculate final coordinates of player, heading towards ball
5      if (on ball) {
6          Calculate random ball location if player on ball
7          MPI_Send [(player coord), (ball coords)] to field process
8      } else {
9          MPI_Send [(player coord)] to field process
10     }
11 }
```

Figure 2.2: Sketch of implementation for a player process (`player_training.c`)

The player processes mainly deal with the calculation of the movement of the respective players in the field. Thus, the only computational heavy section of the processes is the 'movement' calculation.

### 2.3.1 Player's Movement

The player processes are designed such that it runs towards the ball. This is done by keeping track of the 'energy' the player has left and trying to move forward to the ball by moving in the long side first before moving on the short side. If the player does not have enough energy to move to the next cell, then it will utilise the current location as the destination.

## 2.4 Field $\leftrightarrow$ Player Communication

In this section, we will investigate the communication between the field process and the player processes. All point-to-point communication in the program is asynchronous & blocking. As all processes cannot continue execution till it has received the relevant information, non-blocking communication would not be ideal.

### 2.4.1 Rank to Process Mapping

In our program, we make the process with rank 0 as the field process. All player processes have ranks 1 to 11. In this implementation, the rank of the players does not have any significance.

### 2.4.2 Ball Phase

In this phase, the field process sends the ball information (as a `coordinates` struct) to all player processes in a systematic fashion: consecutive, increasing ranks. The sending order does not matter as all player processes, at this point of time, would have already sent their updated location (if in between two games) and are awaiting on `MPI_Send`.

### 2.4.3 Update Phase

Unlike in the previous phase, the order in which the field process receives the coordinates from the players in this phase is important. This is because some processes might not be ready to send over the new coordinates. As such, the field process executes `MPI_Recv` on `MPI_ANY_SOURCE`, so as to receive information form processes that are ready to send.

On top of that, as `MPI_Recv` expects a fixed size message, the ball coordinates will be ignored by the player/field if the player is not on the ball, despite it being sent.

# Chapter 3

# Match – Collective Communication

We have investigated an implementation of MPI football using point-to-point communication. In this chapter, we will be looking into how MPI football (more specifically, the match version) is implemented solely using collective communication between processes. Relevant files are located in either: `common`, `headers`, `match` folders.

## 3.1 Overview

As with the previous chapter, this program can be divided into several phases:

- Ball knowledge phase
- Player movement and challenge phase
- Field '0' (master) state update phase
- Ball possession phase
- Master state printing phase

We will now investigate certain aspects (excluding communication, which will be covered in the next section) of both the field and player processes that are particularly noteworthy.

## 3.2 Fields

The code for the field process is defined in `match/field_match.c`.

```
1  buffer ← player packet array of size 23 or 276 (field 0)
2  set player's initial coordinates
3  while (there exists rounds) {
4      if (is FP0) {
5          check if ball is kicked
6      }
7      MPI_Bcast ball coordinates to world
8      MPI_Gather on field-player communicator, itself as the destination
9      MPI_Gather on field-only communicator, with FP0 as destination
10     if (is FP0) {
11         Determine `winner' of ball
12         Print round statistics
13     }
14 }
```

Figure 3.1: Sketch of implementation for a player process (`field_match.c`)

### 3.2.1 Player's State

The state of the players is not actively stored by all other field process (with the exception of master), unlike in the training program. This allows us to only maintain a single array of player information and override the array on subsequent rounds on all other field processes.

### 3.2.2 Size of Player Packet Buffer

Even though there are only 22 players, the size of the buffer for communication is 23. This is because collective communication would also collect data from the receiving process, even though in our case, the receiving process does not need to 'contribute' data. As such, this data would just be ignored and it is only there to satisfy requirements for MPI collective communication functions.

### 3.2.3 Ball Challenge Calculation

In order to reduce the complexity of the communication, we made it such that the master field process does the tabulation of who 'wins' the ball. After all, this same process needs to print out the same data.

If this was done on the field process that was in charge, it would mean that the 'Master State Update' phase would involve a differently structured packet. Such difference in structure would mean that the existing values in the Player $\rightarrow$ Field gather would have to be converted, which would increase the execution time.

As the master field process also has to compact the array gathered from all field processes, it would be an ideal location to add in the ball challenge calculation, with minimal increase in execution time.

## 3.3 Players

In this section, we will investigate how the player process behaves in this program.

```
1 set player's initial coordinates
2 set player's attributes
3 while (there exists rounds) {
4     MPI_Bcast ball coordinates from field
5     Performs calculation of next position, ball location and challenge
6     MPI_Gather on all field-player communicators (valid only for one of them)
7 }
```

Figure 3.2: Sketch of implementation for a player process (`player_match.c`)

### 3.3.1 Attributes

The player attributes are distributed randomly, as it would allow us to see some variance between matches. While it may be possible to have more deterministic allocation of attributes based on the player's position, it seems that there is an inherent flaw in that an 'all-rounded' player (with average dribbling and speed) would be in possession of the ball most of the time, thus preventing other players form obtaining the ball.

### 3.3.2 Strategy

The playing strategy utilised is a variant of the training phase strategy. While the players still try to rush towards the ball, the players exercise some sort of 'positional discipline'. As such, if the players cannot reach the ball within 2 rounds, the player will not attempt to rush towards the ball. This would prevent 'clumping' of players around the ball and make the game more fairer to players with less dribbling skill.

## 3.4 Field & Player Communication

In this section, we will be investigating the communication between processes in our match. All collective communications are blocking, as it requires participation from all processes that are involved.

### 3.4.1 Rank Mapping

The mapping of the field processes to the MPI ranks have been established in the problem description. However, we further designate field process 0 as the 'master' process, which will perform additional tasks (for instance, printing). In the case of the players, we will map ranks 12 onwards as follows:

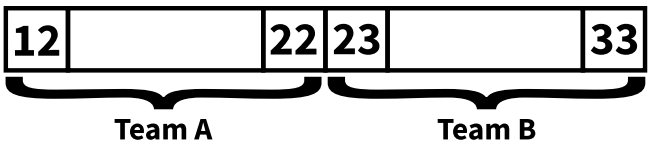


Figure 3.3: Mapping of remaining processes

This allows us to partition the players without having to require a variable/state to be kept in each player process, as this method is deterministic.

### 3.4.2 Communicators

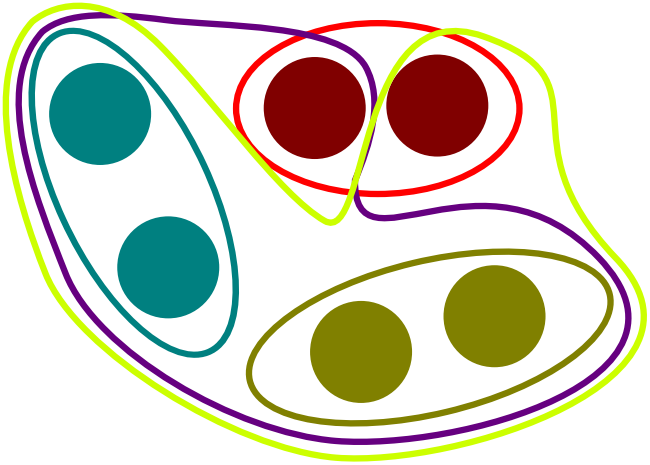

Figure 3.4: Simplified diagram of communicators in the MPI program[1]

In this program, there are a total of 14 communicators (excluding `MPI_COMM_WORLD`), which can be split as follows:

- 1 Field communicator (diagram: red) – all field processes
- 2 Team communicators (diagram: blue and gold) – one communicator each for all player processes in the same team
- 12 Field - Player communicators (diagram: green and purple) – to facilitate communication between players and field. In this communicator, rank 0 denotes the field process.

### 3.4.3 Ball Phase

In this phase, a collective broadcast operation is utilised to transmit the ball coordinates from the master process to all other processes (field and player). This operation is chosen as the type of communication desired is a one-to-many.

[1]Process are coloured like so: red (fields), blue (team A) and gold (team B)

### 3.4.4 Player Update Phase

In this phase, the player process determines the final position and sends in the updated position, ball challenge and a proposed location for the ball. As there is a constant number of information to be sent and that MPI collective communication gathering operations such as Reduce/Gather requires that all processes send the same amount of information, we shall structure the information that needs to be sent as a packet object.

#### Structure of Packet

```
1 typedef struct plypkt {
2     unsigned char isvalid;
3     coordinates position;
4     coordinates ballpos;
5     short challenge;
6 } playerpacket;
```

Figure 3.5: Structure of communication packet

As we can see in the above figure, the structure of the packet is a basic C struct. There are several alternatives that can be used to structure the packet. They are: **bit-fielded/union-ed structs** and **bit masking** a large C data type (e.g. `long`). However, these alternatives were not pursued because of the multi-endian nature of the three-system lab setup. While the i5 and i7 machines utilises little endian, the Jetson board has a bi-endian processor. Since C does not provide any specification on how a bit-fielded struct is laid out, having a heterogeneous setup (like the three machines) would mean that each machine is free to have different ways of arranging the information in a bit-field or union struct.

This also explains why `char` is not used for the `challenge` attribute, as the Jetson board interprets `-1` sent by another machine as a large negative number (and vice versa) due to endianness. Thus, size of the packet is sacrificed to ensure compatibility between different types of systems.

#### Gathering the Packet

The player processes execute `MPI_Gather` for every player $\leftrightarrow$ field communicator (12 in total), indicating that rank 0 (the field) is the receiving process. The player process will mark the packet as valid if the communicator is for the field the player will be on.

On the field processes, every field process would execute `MPI_Gather` on the respective player $\leftrightarrow$ field communicator that it is currently on. This does not require coordination between field processes, as all gather operations executed by the fields are on mutually exclusive communicators. Each field will expect exactly 22 packets, one from each player process, irrespective of the validity of the packets themselves. This is due to the fact that, as discussed previously, `MPI_Gather` requires each 'sending process' to contribute a fixed amount of data.

### 3.4.5 Master Update Phase

In this phase, each field will execute `MPI_Gather` on the field-only communicator, with rank 0 (i.e. field process 0) as the receiver. Thus, this means that field process 0 would need an array of player packets with size $12 \times 23 = 276$, as again, we can't have messages that vary in size across different processes.

#### Possible Alternative: `MPI_Reduce`

If the program was to be run on a heterogeneous system (say, NSCC ASPIRE), the structure of the communication packet can be made such that `MPI_Reduce` can be utilised (with let's say, `MPI_MAX` reducing function). One such example is to use a bit-masked `long`. This would allow us to only require an array of size 22 on the master process, as only the 'valid' packet (i.e. MSB bit set as 1) would only be considered for each player. This also exploits the fact that `MPI_Reduce` reduces each item in an array separately from each other.

The reason why this was not implemented is due to the bi-endian nature of the Jetson machine. Thus, the MSB might not necessarily be the 'valid' flag.

## 3.5 Implementation on NSCC

This program can be implemented on the National Supercomputing Center (NSCC) ASPIRE 1 cluster. In fact, there is no change in the code and it's actually easier to set up the program. In this setup, there's no need for a machinefile as PBS software in NSCC would handle the mapping of processes to machines in the cluster. One needs to modify the file `match.pbs` to change the scale of the machines/processes/cores.

The program that will be created by the `makefile.nscc` is named `matchon.nscc`.

# Chapter 4

# Results & Analysis

In this chapter, we will be performing the execution of the MPI programs and analyse/rationalise the results that were obtained by executing `runfile.lab`, which basically runs all experiments (once) and measures the execution time using `perf`. As is the standard, the experiment is run 5 times and the best result (i.e. lowest execution time) is picked, as this reduces the percentage of time that is not spent on the program.

## 4.1 Point-to-Point Communication

### 4.1.1 Results

| Machine | Execution Time (s) |
|---|---|
| i5 only | 1.3740 |
| i7 only | 1.1581 |
| Jetson | 1.8064 |
| i5 + i7 | 1.6505 |
| All three | 2.1503 |

Table 4.1: Execution times for `training_mpi`

### 4.1.2 Analysis

We can analyse the aforementioned results in two ways: between single machines and between single and multi machine runs.

#### Single Machine Comparison

In the three executions on only one machine in the three machine cluster, we can observe that the Jetson has the slowest runtime, i5 machine being the second slowest and the fastest, the i7 machine. This is expected, as it corresponds to the number of logical cores in the machines, the i7 machine having 8 logical cores (4 physical cores running 2 threads, HyperThreading, each), as opposed to the i5 and Jetson, each having only 4 cores. Thus, this means that the i7 machine is able to execute more processes in parallel at any given time.

Between the i5 and the Jetson, while both have the same number of logical cores, the fact that the i5 is a desktop CPU, as opposed to the more mobile, low power ARM Cortex on the Jetson, means that the i5 is able to execute the same number of processes faster than the Jetson, despite having the same number of processes running at one time.

#### Single vs Multiple Machine Comparison

It seems that the execution time on multiple machines ($T_m$) can be roughly modeled using the execution times for the constituent single machines ($T_{single}$) in the multi-machine setup, like so[1]:

[1]Note that this assumes negligible communication in the single versions

$$T_m = \text{MAX}(T_{single}) + T_o \tag{4.1}$$

The first part of the equation tells us that the multi-machine execution largely depends on the slowest 'machine'. As the single machine execution times only includes negligible Core – Core communication costs, the time measured is highly reflective of its actual raw execution time. Thus, this implies that the machine that is the slowest to calculate (presumably player movements) affects the multi-machine execution time. This is probable as a round only ends when all processes (except the field process) have sent the new position to the field. Thus, the field process has to wait for at most the 'slowest process' before it can proceed on.

The second part demonstrates the latency between Core – Core communication (in single machine) and Machine – Machine communication. As all three machines are connected via Ethernet, this connection introduces larger delay, as the processes are now physically 'further' apart, as opposed to the Core – Core communication on single machines, which happens between processes that are physically closer.

## 4.2 Collective Communication

### 4.2.1 Results

| Machine | Execution Time (s) |
|---|---|
| i5 only | 5.8519 |
| i7 only | 3.6508 |
| Jetson | 17.6141 |
| i5 + i7 | 5.5266 |
| All three | 18.0805 |

Table 4.2: Execution times for `training_mpi`

### 4.2.2 Analysis

#### Single Machine Comparison

As with the point-to-point communication, the execution times for single machines is as expected due to the same reasons that is laid out in the point-to-point section of this analysis. Due to the differences between the point-to-point and collective versions of MPI Football, a fair comparison cannot be made as to whether point-to-point or collective is 'faster' on a single machine.

However, we can suggest that the runtime for collective will be longer than point-to-point, as collective requires all process to reach the same point. Thus, the execution time is strongly dictated by the execution time (excluding communications) of the slowest process, as it must finish too before the collective communication takes place. This is unlike point-to-point, where the communication between other processes can take place in parallel with a process that might still be evaluating results.

#### Single vs Multiple Machine Comparison

While at first glance, the execution time for multiple machines has a similar model to that of the point-to-point version, it seems that the $T_o$ might be smaller than the point-to-point version. This is because in the collective communication version, the communication overheads is greatly reduced, as the communication all happens at the same time.

While this may saturate connections on very large setups (multiple machines), in our case, it is either not saturated as much (due to the fact that a large number may still be running on the same computer) or that the saturation causes minimal increase in execution time.

Thus, this means that we can simplify the model to as such: $T_m = \text{MAX}(T_{single})$. However, be warned that this is only the case for this setup and not large ones where the latency might be significant (e.g. latency from one end of room to another).

## 4.3 NSCC Setup

As it is hard to profile the program execution on NSCC, we shall use the reported wallclock time as the execution time for the program.

### 4.3.1 Result

The execution takes around 4 till 6 seconds in total.

### 4.3.2 Analysis

The execution time of the NSCC setup is around the same as the i5 and i7 execution times.