DSSC - HPC - Parallel Programming

Assignment 3

Patrick Indri

May 3, 2019

1 Introduction

The aim of this assignment is to solve two exercises concerning MPI programming. The work is organized into two folders (`/Ex_4` and `/Ex_5`), one for each exercise.

2 Exercise 4

The first of the two exercises required to implement a cyclic sum between processors. The processors should communicate in a ring-like fashion (please refer to the slides provided during class).

The folder `/Ex_4` contains a `makefile` to compile the code and a script (`run.sh`) to compile and run it. The script and the executable both take the number of processors as argument.

The processors exchange vectors of integers of size `SIZE` and each processor contains three vectors of size `SIZE`: `X`, `Xright`, and `sum`. At first, `X` is initialised to the rank of the processor, and `sum`, to zero. At each step (there are `npes` steps to complete the ring, where `npes` is the number of processors) a processor sends its content `X` to the processor on its left, which will store it in `Xright`. Each processor then updates its `sum`, adding the content it received from its right, and overwrites `X` with `Xright` to propagate the process. At the end of the cycle, each processor will have:

$$\texttt{sum} = \sum_{rank=0}^{npes-1} rank$$

The communication is performed by means of a non-blocking `MPI_Isend`, which allows a processor to start receiving without waiting to complete its send (the receiving, on the other hand, must be blocking, since the update of `sum` requires all the data from the right processor). An `MPI_Wait` call then waits fro the `MPI_Isend` call to be completed before updating the content of the processor to propagate the sum.
The execution time is measured using `MPI_Wtime`. Its call is preceded by an `MPI_Barrier` to make sure that all the processes are ready to start when the start-time is measured, and that all the processes are done communicating/computing when the end-time is measured.

3 Exercise 5

The second exercise required to initialise a distributed identity matrix of size $N \times N$. The matrix should then be printed (in the correct order) on the standard output if $N < 10$; otherwise, it should be printed on a binary file.

The folder `/Ex_5` contains a `makefile` to compile the code and a script (`run.sh`) which loads the `openmpi` module, compiles the code and runs it. The executable must be fed with the matrix size, passed as an argument. The script, on the other hand, must be provided with the size of the matrix and the number of processors as arguments. In the following example the program will deal with a 10×10 matrix and with 4 processors:

```
$ ./run.sh 10 4
```

If $N \geq 10$ the code writes the identity matrix to the binary file (`output.dat`). The script then converts the binary into a more readable ASCII file (`plain.txt`) by means of `hexdump`. This allows to directly verify the correctness of the code.

The initialisation is performed by striping the matrix over the row index, assigning a portion of the matrix to each processor. If the number of processors is not a divisor of the matrix size, the rest is distributed to all the processors which have `rank < rest`, to guarantee balance. Each processor allocates its portion of the matrix and fills it. Each processor then sends its portion to the master process (process `0`). Depending on N, process `0` prints to the standard output or to file. To avoid allocations, the content of process `0` is overwritten at each `MPI_Recv`. In particular, printing to file is performed using `MPI_File_write`. It should be noted that, even though only processor `0` performs I/O, `MPI_File_open` is a collective routine and it must be called by all the processors.