ItHPC Lab Report

Delft University of Technology

Henrique Dantas, 4172922
H.N.D.M.P.N.Dantas@student.tudelft.nl

February 5, 2014

1. Lab 1: Intro

This lab consists of introductory exercises to MPI. Therefore the answers to each question consist of only of source code, submitted electronically.

2. Lab 2: Poisson's equation

2.1. Part 1

2.1.1. Step 1

It is simple to understand that the program was indeed executed twice since two pairs of statements are written to the terminal in comparison to one pair before the modifications. Since we are now running the same program in two different nodes this behavior is expected.

The result is the following

```
Number of iterations   : 2355
Elapsed processortime : 1.350000 s
Number of iterations   : 2355
Elapsed processortime : 1.360000 s
```

2.1.2. Step 2

After adding the global variable and the necessary call to `MPI_Comm_rank` using the predefined communicator `MPI_COMM_WORLD`.

```
MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
```

The following is printed to the standard output:

```
(1)     Number of iterations   : 2355
(1)     Elapsed processortime : 1.360000 s
(0)     Number of iterations   : 2355
(0)     Elapsed processortime : 1.360000 s
```

2.1.3. Step 3

After rewriting the timing functions mentioned in the exercise description, the new output is as follows:

```
(1)     Number of iterations : 2355
(1)     Elapsed Wtime        : 1.468750 s ( 94.0% CPU)
(0)     Number of iterations : 2355
(0)     Elapsed Wtime        : 1.410156 s ( 97.2% CPU)
```

2.1.4. Step 4

Adjusting the code so each process writes to a separate file does not affect the text displayed, so there is no need to repeat it here. In addition by executing the command

```
diff output0.dat output1.dat
```

I was able to confirm the files are indeed identical.

2.1.5. Step 5

On this step, responsible to ensure correct distribution of information originated from an input file, several statements had to be rewritten. Below is a summary of those changes, in particular the parts that were not completely specified in the exercise manual.

To ensure only process 0 opens the file a simple comparison suffices

```
/* only process 0 may execute this if */
if (proc_rank == 0)
{ ... }
```

To broadcast the data read from the file it is first necessary to explain which fields the `MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm))` function requires.

For our situation the `buffer` pointer should refer to the address of the variable we want to broadcast. The `count` relates to the number of entries in the buffer. The `datatype` should describe the type of data the buffer points to, *e.g.* for integers this should be `MPI_INT`. The `root` is the message broadcaster, in our case node 0. Finally we will use the usual predefined communicator for the last argument `comm`.

Thus the broadcast calls are as follows

```
    /* broadcast the array gridsize in one call */
    MPI_Bcast(&gridsize       , 2, MPI_INT    , 0, MPI_COMM_WORLD);
    /* broadcast precision_goal */
    MPI_Bcast(&precision_goal , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* broadcast max_iter */
    MPI_Bcast(&max_iter       , 1, MPI_INT    , 0, MPI_COMM_WORLD);
(...)
    /* The return value of this scan is broadcast even though it is no
    MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
(...)
    /* broadcast source_x */
    MPI_Bcast(&source_x  , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* broadcast source_y */
    MPI_Bcast(&source_y  , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    /* broadcast source_val */
    MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

2.1.6. Step 6

Following the same approach as in the previous section, only the finished version of incomplete code from the manual will be shown in the excerpt.

```
MPI_Comm_size(MPI_COMM_WORLD, &P);
(...)
MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around, reorder, &
(...)
/* Rank of process in new communicator */
MPI_Comm_rank(grid_comm, &proc_rank);
/* Coordinates of process in new communicator */
MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
(...)
/* rank of processes proc_top and proc_bottom */
MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom);
/* rank of processes proc_left and proc_right */
MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right);
```

There a couple new function calls on this code whose arguments I will explain next. As explained in the exercise description MPI uses the `MPI_Cart_*` function calls to arrange tasks in a virtual process grid.

To create one the API calls needs the previous communicator, in our case we were using `MPI_COMM_WORLD`. `ndims` and `dims` define the number of dimensions of the grid and the number of processors in each, respectively. For our example these are 2 and `P_grid`. Thereafter the periods specifies if the grid is periodic or not per dimension, and finally if the ranking is reordered or not. These are replaced by the self-explanatory variables `wrap_around` and `reorder`. The new communicator is stored in the address of `comm_cart`. From now on all pointers to communicators refer to this one.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *perio
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source
```

To define the coordinates of the process in the new communicator we use `MPI_Cart_coords`. Here the rank is necessarily the processor rank (`proc_rank`), the maximum dimensions is 2. The coordinates of each specified process are stored in the `proc_coord` array.

Finally the shift operation returns the shifted source and destination ranks. The direction and displacement quantity arguments are self-evident and are replaced in the program by `X_DIR` or `Y_DIR` and 1 respectively for horizontal and vertical displacements. In accordance to the outputs source

and destination are stored in `proc_top` and `proc_bottom` or `proc_left` and `proc_right` depending on the direction.

The text written to the standard output now features the coordinate of each processor.

```
(1) (x,y)=(1,0)
(1)     Number of iterations : 2355
(1) Elapsed Wtime:        1.464844 s ( 95.6% CPU)
(0) (x,y)=(0,0)
(0)     Number of iterations : 2355
(0) Elapsed Wtime:        1.414062 s ( 96.9% CPU)
```

As one can observe since there are two processor, number 0 is allocated the left half of the grid, and processor with rank 1 deals with the right half.

2.1.7. Step 7

After adjusting the `Setup_Grid()` function as described the following results were obtained for three different setups. The first has two processors, the second three and the third four. The results can be seen below.

```
(1) (x,y)=(1,0)
(1)     Number of iterations : 945
(1) Elapsed Wtime:        0.347656 s ( 83.4% CPU)
(0) (x,y)=(0,0)
(0)     Number of iterations : 1195
(0) Elapsed Wtime:        0.394531 s ( 86.2% CPU)

(1) (x,y)=(1,0)
(1)     Number of iterations : 695
(1) Elapsed Wtime:        0.214844 s ( 79.1% CPU)
(2) (x,y)=(2,0)
(2)     Number of iterations : 1
(2) Elapsed Wtime:        0.046875 s (106.7% CPU)
(0) (x,y)=(0,0)
(0)     Number of iterations : 1
(0) Elapsed Wtime:        0.007812 s (  0.0% CPU)

(2) (x,y)=(1,0)
(2)     Number of iterations : 1
(2) Elapsed Wtime:        0.058594 s ( 68.3% CPU)
(1) (x,y)=(0,1)
(1)     Number of iterations : 792
```

```
(1) Elapsed Wtime:        0.160156 s ( 99.9% CPU)
(3) (x,y)=(1,1)
(3)     Number of iterations : 791
(3) Elapsed Wtime:        0.164062 s ( 97.5% CPU)
(0) (x,y)=(0,0)
(0)     Number of iterations : 791
(0) Elapsed Wtime:        0.128906 s ( 85.3% CPU)
```

2.2. Part 2

3. Lab 3: Finite

4. Lab 4: Nbody

5. Lab 5: Matmul