ItHPC Lab Report

Delft University of Technology

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

February 19, 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:

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

2.1.3. Step 3

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 *peri
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 *sourc
```

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.

```
(0) Number of iterations : 2355
(0) (x,y)=(0,0)
```

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

When there are three processors the work can not be evenly split between them. This can be confirmed by inspecting the `x` and `y` variables in the `Setup_grid function`.

For example for processor 2 (in a 3 processor configuration) it is visible that `x` is always negative.

```
(2) x = −30, dim[X_DIR] = 36
(2) y =  71,  dim[Y_DIR] = 102

(2) x = −3, dim[X_DIR] = 36
(2) y = 76, dim[Y_DIR] = 102

(2) x = −28, dim[X_DIR] = 36
(2) y =  26, dim[Y_DIR] = 102
```

2.1.8. Step 8

2.1.9. Step 9

After implementing the collective reduction operation the total number of iterations is indeed the same, as confirmed by the program's output.

```
(0) Number of iterations : 2355
(0) Elapsed Wtime:         1.507812 s ( 96.2% CPU)
(1) Number of iterations : 2355
(1) Elapsed Wtime:         1.496094 s ( 97.6% CPU)
(2) Number of iterations : 2355
(2) Elapsed Wtime:         1.507812 s ( 96.8% CPU)
```

2.1.10. Step 10

2.2. Part 2

2.2.1. 2.1

2.2.2. 2.2

After changing the code to accommodate for the algorithmic improvement, 5 tests were performed to compare different values for the relaxation parameter ω. The results are summarized in table 2.1. From this data we concluded that 1.93 is the optimal value for the relaxation parameter, accomplishing almost 18 times less iterations than the original with ω equal to one.

ω	`Wtime`[1] (s)	n	Reduction
1.00	1.500000	2355	1.00
1.90	0.222656	220	10.7
1.92	0.199219	165	14.3
1.93	**0.175781**	**131**	**18.0**
1.94	0.289062	142	16.6
1.98	0.351562	419	5.62

Table 2.1: Time, number of iterations obtained and respective iteration reduction for different ω values. The topology used was `pt:`441 with a grid size of `g:`100x100.

2.2.3. 2.3

The goal of this exercise is to investigate the scaling behavior of the code with a fixed relaxation parameter. To accomplish that analysis several runs were measured with various grid sizes. In addition different *slices* were also tested.

[1]This value was computed as the maximum `Wtime` over the four individual processor times for each ω value.

		`Wtime` (s)		
Grid Size	n	`pt:`441	`pt:`422	Δ (%)
200	50	0.164	0.164	0.00
	100	0.226	0.246	8.62
	200	0.343	0.335	-2.27
	300	0.464	0.449	-3.36
400	100	0.437	0.460	5.36
	300	1.054	1.054	0.00
	500	1.746	1.667	-4.47
	1000	3.308	3.304	-0.12
800	100	1.414	1.445	2.21
	300	3.742	3.726	-0.42
	500	5.976	5.988	0.20
	1000	11.625	11.617	-0.07
2000	100	8.218	8.234	0.19
	300	21.351	21.328	-0.11
	500	34.933	35.492	1.60
	1000	68.304	67.824	-0.70

Table 2.2: The maximum time for different grid sizes and different *slicing* arrangements. The Δ column compares the relative difference in performance between the two previous columns. The number of iterations for each run was fixed to enable a more accurate comparison. The 200x200 grid size measurements were performed in different conditions in regards to the iteration count since the program converges after 382 iterations.

The results of the aforementioned experiment are shown in table 2.2. As one can obverse the different domain partitions have little impact in the overall performance of the program, even as the grid size increases.

As explained in the exercise manual the (average) time t per iteration n can be parametrized as follows

$$t(n) = \alpha + \beta \cdot n \tag{2.1}$$

Where α and β are arbitrary constants. To determine these constants we will use the least-squares method. The results of applying this technique[2] to the available dataset are shown in table 2.3

	`pt`:441		`pt`:422	
Grid Size	α	β	α	β
200	0.105	0.001	0.120	0.001
400	0.116	0.003	0.115	0.003
800	0.310	0.011	0.329	0.011
2000	1.449	0.067	1.721	0.066

Table 2.3: Using least-squares method we estimated α and β as defined in equation 2.1. The dataset adopted for this computation is present in table 2.2.

In order to improve the quality of this an estimation a thorougher study should be performed with more runs. In addition considering other (non-linear) parametrization functions, *e.g.* exponential, may also yield interesting results.

To conclude this question we plotted the estimated functions along with the actual data points to facilitate a visual comparison of the estimation.

[2]The Google Drive implementation of least-squares, *LINEST* function, was used for this computation. For more information please see `https://support.google.com/drive/answer/3094249`.

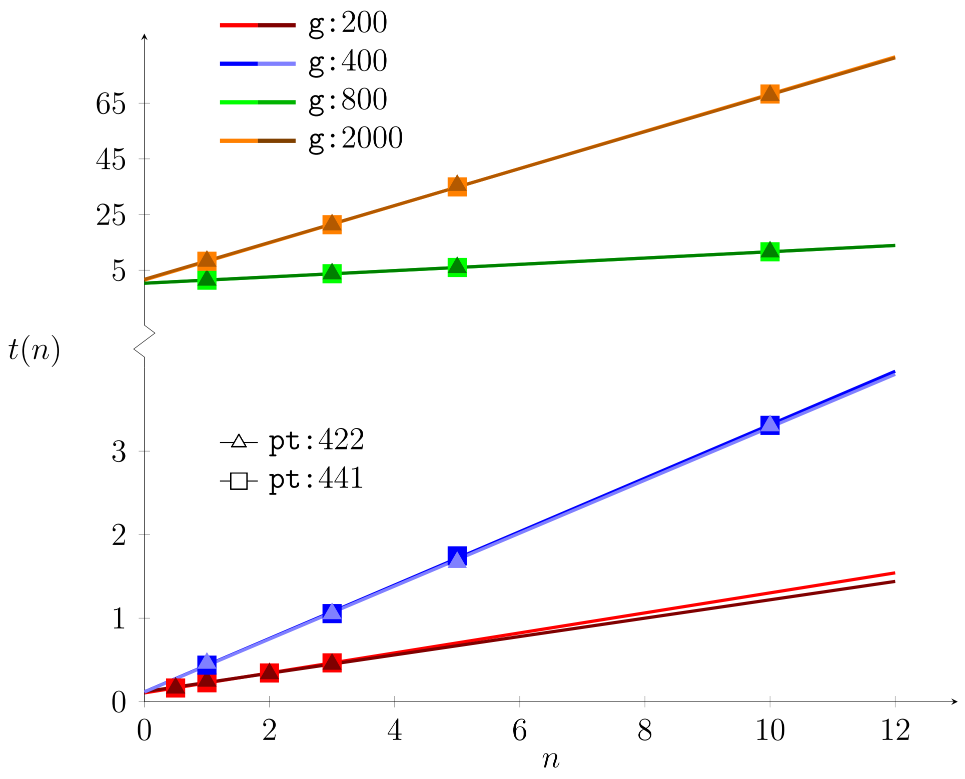

Figure 2.1: Visual comparison between the experimental data and respective linear estimation using the least-squares method.

As figure 2.1 shows the estimated α and β can very accurately determine the time necessary to complete the operation. This can be observed by noticing that the measured data points (depicted as -△- and -□- for the two different partitions) are located close to the curves (which represent the estimated values).

3. Lab 3: Finite

4. Lab 4: Nbody

5. Lab 5: Matmul