Assignment One – C and MPI (Group Work Allowed, MAX 2 People)

Rita Borgo

October 29, 2014

Number of Credits: 30% of a 10 credit module.
Recommended hours: approximately 30 hours.
Submission Deadline: Wednesday 19 November 2014 by 11am.

The Coursework can be developed in groups of maximum two members (no larger groups allowed), the work can be carried out individually as well.

1 Coding Convention [5%]

If you write code in any language as part of a larger project, you will be expected to conform to a set of coding conventions. 5% of the marks in Coursework 1 are awarded for code conforming to the set of rules described in Appendix A.

2 C Assignment [50%]

For this assignment, you are required to implement a time dependent temperature distribution model. The model is a 2D version of the one seen in class.

2.1 Problem Description

Let us suppose we know the temperature distribution of a specific region at time t, we want to know how the temperature distribution will look like at time $t + \delta t$. Assuming temperature changes through time can be modeled according to the mathematical model described in equation 1, first we discretise our domain into $N \times N$ equal size grid cells of size δx and δy, as shown in in Figure 1. For each point $u_{i,j}$, in our 2D grid at time t, we compute the corresponding value at time $t + \delta t$ using the following equation:

$$U_{x,y,t} = U_{x,y,t-1} + C_x \times (U_{x+1,y,t-1} + U_{x-1,y,t-1} - 2 \times U_{x,y,t-1}) + C_y \times (U_{x,y+1,t-1} + U_{x,y-1,t-1} - 2 \times U_{x,y,t-1}) \tag{1}$$

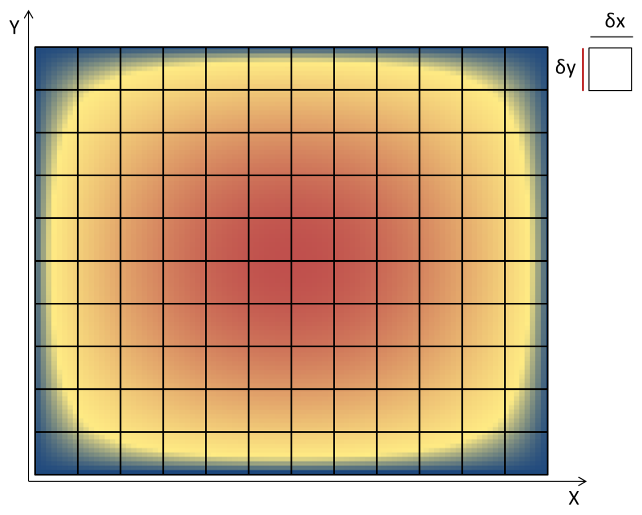

Figure 1: Domain decomposition. Temperature distribution at timestep $t = 0$.

2.2 Program Description

You should write a C program that implements equation 1. A file, `serial_temp_2D.c`, is provided containing the skeleton of a possible solution (you can write your own code if you prefer as long as it meets the specified requirements). The file does not compile as it is and it is your task to complete all missing sections according to the following instructions:

Instructions. In writing this program you should use two two-dimensional arrays of doubles:

```c
double new_u[XDIM][YDIM], old_u[XDIM][YDIM];
```

To simplify our task we keep the arrays fixed in size, with XDIM=YDIM=100 elements.

The array new_u will store the temperatures that are being currently computed (i.e.,$u(x_i, y_i, t_{j+1})$) and old_u will store the temperatures corresponding to the previous timestep.

You should also use a structure (structure `Params` in the example code) to store parameters C_x, C_y and **nts** passed at command line.

Your program shall:

- read in C_x;
- read in C_y;
- read in **nts** - the total number of timesteps to be computed;
- read the initial grid conditions - the values of our grid at time $t = 0$, passed as a .csv file;
- initialise the parameter structure;
- initialise `old_u` with the initial grid conditions;
- for each timestep from $t = 1$ to timestep $t = nts$ compute new grid values and store them in `new_u`;

To simplify our task we assume that values at the boundaries of our region remain unchanged through time, we call these our **boundary conditions**. This means that through each time steps values at positions:

$$\begin{array}{lll} u[0][j] & with & 0 \leq j \leq YDIM - 1 \\ u[i][0] & with & 0 \leq i \leq XDIM - 1 \\ u[XDIM-1][j] & with & 0 \leq j \leq YDIM - 1 \\ u[i][YDIM-1] & with & 0 \leq i \leq XDIM - 1 \end{array} \tag{2}$$

remain unchanged and can be copied directly from timestep t to timestep $t + \delta t$

Testing Instructions You shall rename your executable to `temp_dist2D`.
Your program must compile on the Linux Workstation in the Linux Lab. room 217L, Faraday Building. The executable should run at command line as follows:

```
./temp_dist2D 0.1 0.1 50 initial_data.csv
```

The command line arguments would denote a value of C_x of 0.1, a value of C_y of 0.1, a total of 50 timesteps to be computed and file initial_data.csv containing grid values at timestep $t = 0$.
An initial testing file (initial_data.csv) has been provided, you can test your program with the input suggested above (e.g. `0.1 0.1 50 initial_data.csv`).

2.3 Part 2

Basic Performance Assessment-Experiment. Try to change the number of time steps and time the results (lookup `time.h`). Answer the following questions in a clearly defined comment at the top of your C code.

- **Question 1** How does the algorithm scale when increasing the number of time steps?
- **Question 2** Do you hit a limit at any point (e.g. computational time > than a few minutes or local space not sufficient)?

Saving intermediate timesteps. The provided skeleton code only saves the final timestep data into file *final_data.csv*. Modify the printing section of the code such that every newly generated timestep dataset is saved in a file named *final_data_ti.csv* with *ti* being the computed timestep, e.g:

final_data_1.csv correspond to data generated at timestep 1

final_data_2.csv correspond to data generated at timestep 2

...

final_data_49.csv correspond to data generated at timestep 49

Extra credits: in the code provided `update` function is called twice, first as

$$update(dim, dim, \&old_u[0][0], \&new_u[0][0]); \tag{3}$$

and second as

$$update(dim, dim, \&new_u[0][0], \&old_u[0][0]); \tag{4}$$

can you explain why we swap `old_u` with `new_u` ? Add your answer in a clearly defined comment at the top of your C code.

3 MPI Assignment [45%]

Build the merrygoround.c example using mpicc, run your example in the Linux Lab:

- Try to understand how this code works.
- Compile, then run the executable using 1, 4, 8 and 16 nodes and an appropriate machinefile.
- Change the number of megabytes sent in the code, rebuild, and rerun on 1, 4, 8 and 16 nodes.
- Put the output from the above in a table with number of nodes as rows and number of megabytes as columns.
- Create a graph plotting the results collected in your table (you can use Excel or any other graphics package of your choice) and discuss your results in terms of speedup.
- Place the output from the above and your discussion in a file named <studentNumber>_A1-MPI.pdf.
- Place the .pdf file and the machinefiles used for each test in a folder named <studentNumber>_A1-MPI.

4 Submission Instructions

4.1 Platform

For this assignment your code must compile and run on the Linux machines in room 217L Faraday Building.

4.2 Submission

Submission via Blackboard follows the normal procedure (if you submit as a group only one member of the group will need to perform the submission). Create and submit an archive file labeled as:

- if submitted as individual work: <studentNumber>_A1.zip
- if submitted as group work: <studentNumber1_studentNumber2>_A1.zip.

The archive file shall contain the following items:

1. A folder named CPROG_A1 containing the code implementing Section 2 (file .c, .h optional) and the ouptut (file .csv) created with the suggested input parameters (e.g. `100 0.1 50 initial_data.csv`).
2. A folder named MPIEX_A1 with the results from the Experiment in Section 3.

All submitted material must be original work.
Note: You will be notified in lectures when marking has been completed if you have not received a mark you must notify me immediately.

A Appendix A Coding Conventions

A.1 Indenting

Each new depth of nested block should be indented by one tab. The braces around each block should be on their own line, at the same indent level as the preceding line. All conditionals and loops should be followed by a new block, even if they only contain one statement.

For example:

```
void function ( void )
{
// Indent is one tab here
	if ( someCondition )
	{
// Indent is two tabs here
		doStuff ();
	}
}
```

A.2 Spacing

Insert blank lines between related groups of statements, to visually group the set of statements. When calling a function, there should be no space between the function name and the call operator, but parenthetical expressions after a language keyword must have a space separating them.

For example:

```
if(x)
{
	if(y)
	{
		doStuff(); // Correct
		thenMoreStuf (); // Wrong !
	}
}
```

There should be no space before the semicolon, or other punctuation, but there should be space after each comma. Spacing within expressions should mirror the operator precedence. When declaring pointers, the * should be adjacent to the variable. When using pointer types in other contexts, it should be adjacent to the type name.

For example:

```
int* example ( void )
{
	int *a;
	a = (int *)b;
}
```

A.3 Comments

Comments should describe why things are being done. All functions and data structures should be declared with documentation comments (starting /**) explaining their purpose.

For example:

```
/**
Structure for storing a point, using 2D Cartesian
coordinates .
*/
struct Point
{
      /** X (horizontal) coordinate. */
      float x;
      /** Y (vertical) coordinate. */
      float y;
};

/**
Determine whether two points intersect . Returns 1 if
they do.
*/
int pointsIntersect ( Point p1, Point p2);
```

A.4 Naming Conventions

Function names should start with a lowercase letter and describe the expected behaviour of the function. Global variables and structure names should begin with an uppercase letter. Type definitions that refer to pointer types should use the _t suffix. Preprocessor macros should be entirely in uppercase.

For example:

```
typedef struct Point *point_t ;
```