Programming Assignment 5 — Due June 5th 12:50PM

In this assignment you will be using MPI to implement the heat diffusion algorithm from Programming Assignment 3. You will learn more about distributed memory models and implementing code for use on a large cluster.

Background
We will be solving the 2D Heat Diffusion equation that we solved in PA3. Recall that the implementation required the following parameters:

```
int    nx_, ny_;     //number of grid points in each dimension
int    gx_, gy_;     //number of grid points including halos
double lx_, ly_;     //extent of physical domain in each dimension
double alpha_;       //thermal conductivity
double dt_;          //timestep
int    iters_;       //number of iterations to do
double dx_, dy_;     //size of grid cell in each dimension
double xcfl_, ycfl_; //cfl numbers in each dimension
int    order_;       //order of discretization
int    borderSize_;  //number of halo points
```

As in PA3, we will use a file `params.in` that we will pass (in the command line) as argument to our program. Specifically, the file `params.in` contains (in this order):

```
int    nx_, ny_;     //number of grid points in each dimension
double lx_, ly_;     //extent of physical domain in each dimension
double alpha_;       //thermal conductivity
int    iters_;       //number of iterations to do
int    order_;       //order of discretization
bool   blocking_;    //is the communication blocking
```

The parameter that is specific to MPI is `blocking_`. We will explain what this parameter actually means as we explain your tasks for this assignment.

Starter code
The starter code is composed of the following files:

- `simparams.cpp`: this file contains the `simParams` class that keeps track of all the parameters of the problem.
- `grid.cpp`: this file contains the `Grid` class that manages the grid on which we will solve the equation.
- `2dHeat_hmwk.cpp`: this file contains the `main` function. It also contains the code that performs the computation solving the heat equation.
- `params.in`: this is the file containing the input parameters of the problem. You should only modify lines 0 (grid dimensions), 3 (number of iterations), 4 (order), 5 (method of communication).
- `Makefile`: this file will compile the code for you. It uses `mpiCC`, so make sure you follow the instructions of Appendix B to enable MPI on the cluster.

The two files you will need to modify (and submit) are `grid.cpp` and `2dHeat_hmwk.cpp`. The places where you need to work are marked as `TODO`.

The starter code divides the global grid into n smaller grids (where n is the number of processes) that will then be handled to each of the processes. Specifically, we use a 1D decomposition along the y-axis: if the global grid is $n_x \times n_y$ big, we will have n grids of dimension $n_x \times \dfrac{n_y}{n}$ each.

We need to store two versions of the unknowns on the grid, one for the previous step and one for the current. Both the previous and current grids are stored in the same vector and, as a result, the `grid_` member has a size of $2 \times g_x \times g_y$. We then use the proper offset to access the current or the previous grid. This allows easily swapping the data at each step (e.g., current becomes previous). Make sure you understand the organization of the vector as this will be of crucial importance for the communication of the boundaries.

Method of communication

As we have seen in class, MPI offers many different ways of communicating between processes:[1] blocking vs. non-blocking, synchronous vs. standard, buffered vs. ready send.

In our problem, the processes will need to communicate their borders to their neighbors from one iteration to another. You will be implementing two different communication methods to achieve this purpose. The method that the code will be using at running time is given by the parameter `blocking_`:

1. `blocking_ = 0`: we use non-blocking communications between the processes. You should use `Isend` and `Irecv` for this task.
2. `blocking_ = 1`: we use blocking communications. For this task, we want you to use `Sendrecv` (although you can use `Send` and `Recv` for the top and bottom processes).

Question 1

(10 points) For this question only, we assume that periodic boundary conditions are used such that all processes need to send and receive data. No special treatment is then required for processes on the boundary. In that case, discuss the possibility of deadlock in the following scenarios:

1. All processes first send a boundary to its top neighbor using `Send` and then receive a boundary from its bottom neighbor using `Recv`. The send operations use buffers.
2. Same as 1, but the send operations do not use buffers.
3. All processes first receive a boundary from its bottom neighbor using `Recv` and then send a boundary to its top neighbor using `Send`. The send operations use buffers.
4. Same as 3, but the send operations do not use buffers.

Question 2

(10 points) Implement `waitForSends` that waits for all sends to finish.

Question 3

(10 points) Implement `waitForRecvs` that waits for all receives to finish.

Question 4

(20 points) Implement `transferHaloDataNonBlocking` that communicates boundaries between processes using non-blocking functions (`Isend` and `Irecv`).

Question 5

(20 points) Implement `transferHaloDataBlocking` that communicates boundaries between processes using blocking functions. You should use `Sendrecv`.

Question 6

(15 points) Implement `computation` that performs the computation aimed at solving the heat equation.

[1] A brief guide to remind you of the different send and receive modes: `http://www.mcs.anl.gov/research/projects/mpi/sendmode.html`.

Question 7

(10 points) Plot for a grid of size 1024 × 4096, with 400 iterations, an order of 8, and using a non-blocking communication scheme:

- The running time t of the algorithm as a function of the number n of processes.
- The inverse of the efficiency e (we omit a constant factor) defined as:

$$\frac{1}{e} = t(n) \times n$$

  as a function of n. $t(n)$ denotes the running time of the program when using n processes.

Make n vary in the range $[\![1, 12]\!]$. Comment your results.

Question 8

(5 points) What would the plot of the efficiency look like if we assume that the communication time is negligible?

Total number of points: 100

Running the code
After having compiled your source files, you will be running the code using:

```
mpirun -np n 2dHeat params.in
```

This should run your code using `n` processes and save the results to `gridX_final.txt`, where $X \in [\![0, \mathtt{n} - 1]\!]$ is the rank of the calling thread.

If you are curious, you can read about the optional arguments to `mpirun` at:

`http://www.open-mpi.org/doc/v1.4/man1/mpirun.1.php`

A Submission instructions

You should submit one zip file containing a folder named `LastName_FirstName_SUNetID_PA5`. Make sure to have a folder with this name otherwise your files will get mixed up with other students when we unzip your file. This folder should contain:

- `grid.cpp`, `2dHeat_hmwk.cpp`
- A pdf file named `Readme.pdf` containing your answers to the questions.

The zip file should be named `PA5.zip`.

To run (and grade) your code, we will copy your source file in our directory and type (we may change the parameters):

- `make`
- `mpirun -np 4 2dHeat params.in`

B Hardware available for this class

Machines

We will be using the icme-gpu teaching cluster, which you can access with ssh:

```
ssh sunet@icme-gpu1.stanford.edu
```

Compiling

To use `mpiCC` (the MPI compiler we will be using) on the icme cluster, you must copy and paste these lines to your `.bashrc` file (so they will be loaded when you connect on the cluster).

```
module add open64
module add openmpi
```

For the first time, after you changed the `.bashrc`, you need to logout and re-login (so that the changes are taken into account) or source the `.bashrc`.

You can now use `mpiCC` to compile your code.

Running

The cluster uses MOAB job control. The easiest way to run an executable is by using interactive job submission. First enter the command

```
msub -I -l nodes=p:ppn=q
```

where p and q are values that will depend on how you want to run your MPI code (the total number of processors you will be assigned is $n = pq$). Be advised that there are 15 nodes with 12 processors each.

You can then run your executable using `mpirun`. For example:

```
mpirun -np 4 2dHeat params.in
```