Programming Assignment 5

Due May 23, 2012 12:50 PM

In this assignment you will be using MPI to implement the heat diffusion algorithm from Programming Assignment 2. You will learn more about distributed memory models and implementing code for use on a large cluster.

1 Background

Recall the 2-Dimensional Heat Diffusion algorithm from Programming Assignment 2.

The computation requires several parameters, so rather than pass them in at the command line we will compute them from the file `params.in`. Here is a list of parameters that are used:

```
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 ic_;          //uniform initial condition
double xcfl_, ycfl_; //cfl numbers in each dimension
int    order_;       //order of discretization
int    borderSize_;  //number of halo points
double bc[4];        //0 is top, counter-clockwise
```

These parameters define the number of particles in the grid, the rate of diffusion, and the amount of time to model the system. An important parameter is the *order*. This controls how many neighbors we will consider in the computation. With a higher order, we will consider both our immediate neighbors and neighbors further away. You will use the points whose distance along the x and y dimensions are at most halo = order/2 from the current point. We call these points the *stencil*. The distance from the center points changes the *weight* of a point in the computation.

We have added two parameters to `params.in` which control the synchronization and domain decomposition, which we describe in the parallelization section.

2 What we give you

- `2dHeat.cpp` - This is the starter code for your implementation. You should add MPI code here to implement the 2-D Heat Diffusion algorithm using both global and shared memory. You should modify this file.
- `Makefile` - `make` will build the 2dHeat binaries. `make clean` will remove the executable. You should be able to build and run the program when you first download it, however only the host code will run. You do not need to modify this file.
- `params.in` - This file contains some parameters and input to the algorithm. You may modify this file if you would like to test different configurations. You should change the following:
  - The grid size (the first line)
  - The number of iterations (the fourth line)

- The order (the fifth line)
- The synchronization mechanism (the seventh line)
- The data partitioning schema (the eigth line)

These files can be found in `/afs/ir/class/cme213/assignments/pa5` on Leland machines (such as the corn and myth clusters).

3 Parallelizing the code

To parallelize this program in CUDA, we had many threads, and each single threads working on a single value (or a very few number of values) in the matrix.

To parallelize on a cpu, we will instead have few threads, and each single thread will handle a large chunk of the matrix.

We will analyze different strategize for parallelizing the code, and we will ask you to answer some questions in the analysis section.

- First, we must consider how we will partition the data:
  - Create the parallelism in one-dimensional sections: each row in the matrix is a task handled by a thread, dividing the rows as necessary.
  - Create parallelism in two-dimensional sections: Divide the matrix into square regions as a task to be performed by a thread.
- Next, we consider the method of communication
  - Asynchronous communication : In order to diffuse the heat over several iterations, nodes will need to communicate their borders to other threads. We can overlap this communication time with computation that is local to a thread. We will use the asynchronous MPI calls to do this.
  - Synchronous communication : Layer on top of your asynchronous solution a synchronous version which forces the threads to wait for all the communication to finish. Note that because we are using our asynchornous implementation, we are not using synchronous MPI calls.

  This guide does a particularly garish job of highlighting the differences between synchronous and asynchronous send and recv:

  `http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/92-MPI/async.html`

4 Testing

Your implementation should support the following commands:

- `make` : compiles your MPI code using `mpiCC`
- `mpirun -np # <-npernode $> 2dHeat params.in` : After calling `msub -I -l nodes=# :ppn=$` , This should run your code on # nodes and save the results to `gridX_final.txt`, where X is the rank of the calling thread, and ranges from 0 to #, the number of threads to launch. You should only use perfect squares as # values (1, 4, 9, 16, ...). Specifically, make sure this works when both # and $ are 4.

  `-npernode` is an optional argument that lets you control how many processes are launched on each node.

  You can read about the optional arguments to mpirun at:

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

5 Analysis

1. Correctness: 50%

   Make sure your program generates correct results for:

   (a) 1-D decomposition and any number of processors, any grid size, async and sync; if the grid size is too small for a given stencil, output an error and halt. Crashing or bogus results are not correct. Should the decomposition be along rows or columns?

   (b) 2-D decomposition and any square number of processors (1, 4, 9, ...), any grid size, async and sync.

   Unlike previous assignments, you are not given a routine that automatically checks correctness for you. You can add one yourself by comparing against a single processor solution or compare against analytical solutions.

2. Performance: 40%

   Once your program is correct we will conduct performance testing.

   (a) Strong scaling: for a 1000x1000 and 2000x2000 grid, how does each method scale with the number of processors. There should be 4 curves on each plot; 1D Async, 1D sync, 2D Async, 2D sync. Force the MPI system to only choose one processor per node, so that all communication between processors must occur over an interconnect

   (b) Repeat (a), but allow the MPI system to choose as many processors per node as it wants. What impact do you expect this to have? Why?

3. Understanding: 10%

   Because of the way we chose to implement the synchronous routine, we only used asynchronous MPI calls. How would the communication pattern change if we were to use synchronous MPI calls for the synchronous communication?

6 Hardware

We will continue to use icme-gpu1, even though we are now only taking advantage of its CPU capabilities.

We will use the familiar `msub` command to access nodes in the cluster: `msub` You can run MPI code through the command `msub -I nodes=4 :ppn=4`. Here, play with different values for the number of nodes and the number of processors per node (and additionally toggle the inputs to mpirun, as described above) to have the computation run in different distributed patterns.

Be advised that there are 15 nodes with 12 processors each.

If you run your code on a different cluster, make sure to enable JOBNODEMATCHPOLICY in the queue configuration file.