Lab 2

ECSE 420 – Parallel Computing – Fall 2016

Released: October 26, 2016
Due: November 8, 2016 at 11:59 pm

In this lab, you will write code for a drum simulation, parallelize it using MPI, and write a report summarizing your experimental results.

Submit a single zip file with the filename Group_<your group number>_Lab_2.zip, or Group_<your group number>_Lab_2_PYTHON.zip if you are using mpi4py.

Finite Element Music Synthesis

In a "finite element" model, a complex physical object is modelled as a collection of simple objects (finite elements) which interact with each other and behave according to physical laws. It is possible to simulate electromagnetic fields, calculate stresses in the structure of a building, or synthesize the sounds of a musical instrument using finite elements. In this lab, you will synthesize drum sounds using a two-dimensional grid of finite elements.

The particular finite element method we will be using is similar to but slightly different from the ocean simulator we examined in Assignment 2. In this method, each interior element performs the following update at every iteration:

$$u(i,j) := \frac{\rho(u1(i-1,j) + u1(i+1,j) + u1(i,j-1) + u1(i,j+1) - 4u1(i,j)) + 2u1(i,j) - (1-\eta)u2(i,j)}{1+\eta}$$

$$1 \leq i \leq N-2, 1 \leq j \leq N-2$$

where u is the displacement of the element at position (i,j), $u1$ is the displacement of that element at the previous time step, $u2$ is the displacement of that element at the previous previous time step, and η and ρ are constants related to the size of the drum, sampling rate, damping, etc.

Then, we ensure that the boundary conditions are met by updating the side elements:

$$u(0,i) := Gu(1,i)$$

$$u(N-1,i) := Gu(N-2,i)$$

$$u(i,0) := Gu(i,1)$$

$$u(i,N-1) := Gu(i,N-2)$$

$$1 \leq i \leq N-2$$

and then the corner elements:

$$u(0,0) := Gu(1,0)$$

$$u(N-1,0) := Gu(N-2,0)$$

$$u(0,N-1) := Gu(0,N-2)$$

$$u(N-1,N-1) := Gu(N-1,N-2)$$

where G is the boundary gain.

Finally, at the end of the iteration, we set $u2$ equal to $u1$ and $u1$ equal to u.

To simulate a hit on the drum, simply add 1 to u1 at some position. To record the sound of the drum, collect the value of u at some position for each iteration in an array of length T, where T is the number of iterations to perform. For this lab, you should use (N/2, N/2) as the position on the drum for both the hit and the recording.

1. Write code which implements a 4 by 4 finite element grid. Parallelize the computation by assigning each node of this grid to a single process using MPI. The user will provide a single command line argument, T, which specifies the number of iterations to run the simulation. Your code should print u(N/2,N/2) (which in this case is u(2,2)) to the terminal at each iteration.

   The grader will run the following command:

   ```
   mpirun -np 16 ./grid_4_4 <T>
   ```

   (You may assume that the grader will run your code with 16 processes and not with any other number of processes. Also assume that T will be a positive integer.)

   If you are using Python, the grader will instead run:

   ```
   mpirun -np 16 python grid_4_4.py <T>
   ```

2. Using one process per finite element can require an enormous amount of communication and OS overhead, and even in large clusters there may not be enough hardware available to fully parallelize the computation. It is usually better to use a decomposition which assigns multiple elements to each processor.

   Parallelize your code using such a decomposition (with rows, columns, blocks, etc.) and simulate a 512 by 512 grid. In your report, discuss your parallelization scheme and experimental results.

   The grader will run the following command:

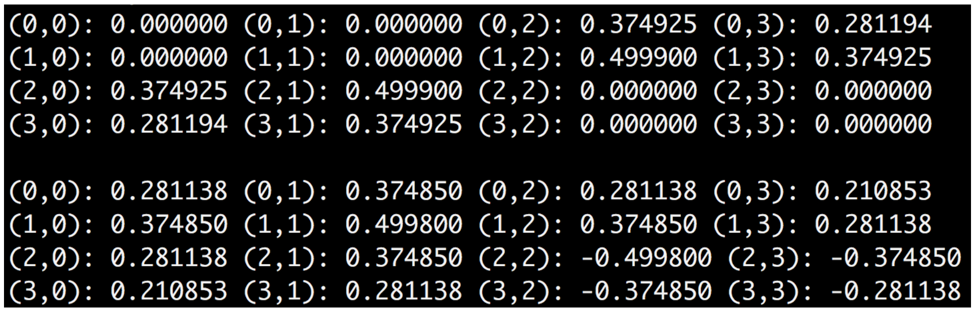
   ```
   mpirun -np <num procs> ./grid_512_512 <T>
   ```

   If you are using Python, the grader will instead run:

   ```
   mpirun -np <num procs> python grid_512_512.py <T>
   ```

   It should be possible for the grader to run your code with 1, 2, 4, 8, 16, or 32 processes.

To help debug your code, Figure 1 shows the entire u grid for two iterations:



```
(0,0): 0.000000 (0,1): 0.000000 (0,2): 0.374925 (0,3): 0.281194
(1,0): 0.000000 (1,1): 0.000000 (1,2): 0.499900 (1,3): 0.374925
(2,0): 0.374925 (2,1): 0.499900 (2,2): 0.000000 (2,3): 0.000000
(3,0): 0.281194 (3,1): 0.374925 (3,2): 0.000000 (3,3): 0.000000

(0,0): 0.281138 (0,1): 0.374850 (0,2): 0.281138 (0,3): 0.210853
(1,0): 0.374850 (1,1): 0.499800 (1,2): 0.374850 (1,3): 0.281138
(2,0): 0.281138 (2,1): 0.374850 (2,2): -0.499800 (2,3): -0.374850
(3,0): 0.210853 (3,1): 0.281138 (3,2): -0.374850 (3,3): -0.281138
```

Figure 1: u for two iterations

(Remember, do not print the entire grid every iteration; just print the value of the output node. This figure is just to help you debug.)

Bonus: Implement the algorithms in both C and Python and compare them in your report. (Hand in only one of your implementations.)

Bonus bonus: Run your code on multiple hosts. Try different host/slot configurations and see which gives you the best speedup. Describe your experimental setup.

When running your code, only use hosts [g, g+1, g+2, g+3] mod 15 + 1, where g is your group number, so that no one host takes on too big a load.

Examples:

Group 1 should only use hosts [2,3,4,5]

Group 20 should only use hosts [6,7,8,9]

Submission Instructions

Your submitted zip file should contain the following:

- a PDF report
- a Makefile (leave empty if you use Python)
- whatever source files you need such that the Makefile produces the correct executables

To test your submission, the grader will unzip your file, enter the unzipped directory, run "make", and then run the commands listed in the sections above. Your code must compile and run on the Trottier Linux machines.

The grading scheme is as follows:

- 4 by 4 grid - 25%
- 512 by 512 grid - 25%
- Report - 50%