CSCI 3366 (Parallel and Distributed Programming), Fall 2019

Homework 4

Credit: 30 points.

1 Reading

Be sure you have read, or at least skimmed, chapters 1 through 5 of the textbook.

2 Overview

Your mission for this assignment is write a parallel version of mathematician John Conway's "Game of Life", as described briefly in class. (You can also find more information on the Web. The Wikipedia article seems good.)

The Game of Life is not so much a game in the usual sense as a set of rules for a cellular automaton: There are no players, and once the initial configuration is established, everything that happens is determined by the game's rules. The game is "played" on a rectangular grid of cells. Some cells are "live" (contain a simulated organism); others are "dead" (empty). At each time step, a new configuration is computed from the old configuration according to the following rules:

- For each cell, we look at its eight neighbors (top, bottom, left, right, and the four diagonal neighbors) and count the number of cells that are live. (Note that this count is based on the configuration at the start of the time step.)
- A dead cell with exactly three live neighbors becomes live; otherwise it stays dead.
- A live cell with two or three live neighbors stays live; otherwise it becomes dead (of isolation or overcrowding).

This problem clearly(?) fits our *Geometric Decomposition* pattern and is fairly straightforward to parallelize. However, it's unlikely that parallelization will improve performance unless the board size is large, and for large boards inputting and displaying (or printing) board configurations gets unwieldy. But it might be interesting to experiment with randomly-generated board configurations and observe how the number of live cells changes over time (does it settle down to a stable number? what and how soon?), so we'll do that.

3 Details

3.1 Sequential program

(5 points)

To help you get started, I wrote a sequential C program with a simple text interface, with command-line arguments that specify:

- input source (an input file or the keyword "random" followed by size, fraction of cells that should initially be "live", and seed)
- number of steps

- print interval (P means to output updated board every P steps)
- optionally, a name for an output file to contain initial and updated boards

The program prints (to standard output) only counts of "live" cells at each step; if an output file is specified, it also writes initial and updated board configurations to it.

The starter code defines data structures, gets input, sets up the board, and writes output, but omits the code that implements the actual algorithm. (Comments with the word "FIXME" show you where you need to make changes/additions.)

- Code: game-of-life.c. You will also need timer.h to compile the program.
- Sample input file: input_8x8.
- Sample output file (using the above input and executing for 4 steps): output_8x8.

Start by filling in the parts of the code I left out and running the result a few times, to test that you understand how to do the computational part of the game.

3.2 Parallel programs

(20 points)

Your next job is to write two parallel versions of this application, one for shared memory using OpenMP and one for distributed memory using MPI. Both parallel programs should produce exactly the same results as the original sequential program, except for timing information.

3.2.1 OpenMP program

This one should be fairly straightforward. As with the OpenMP programs for previous homeworks, have the program get the number of threads to use from environment variable `OMP_NUM_THREADS` and print with the timing information the number of threads used.

3.2.2 MPI program

This one is less straightforward, but doable. As with the MPI programs for previous homeworks, have the program print with the timing information the number of processes used. Suggestions:

- Distribute the "board" among processes so that each process has a block of rows of the original board. (For example, with two processes one process would have the top half of the board and the other the bottom half.) (It's somewhat more customary when distributing a 2D data structure to distribute square blocks, but that makes exchanging boundary information much more complicated, with MPI at least; distributing blocks of rows is simpler.)
- Have only process 0 open the output file and print to it. Other processes can send "their" rows to process 0, which can collect and print them. Since there's no need to do that if there's no output file, you might have process 0 broadcast a value indicating whether there *is* an output file.

3.3 Performance of parallel programs

(5 points)

Once you have working parallel code, experiment with input values until you get a problem size/configuration big enough to make it reasonable to hope for good speedups with multiple UEs. (Think a little about what will affect this most — size of board, number of steps, interval between printing results.) Then time your two parallel programs for this problem size/configuration and different numbers of UEs and plot the results, as in Homework 3.

4 What to turn in and how

Turn in the following:

- Source code for your sequential and parallel programs.
- Results of measuring performance. For each of these programs, tell me what inputs you used for the program, which machine(s) you ran it on, and send me:
  - A plot showing how execution time depends on number of UEs.
  - Input data for the plot. A text file or files is fine for this.

Submit your program source code by sending mail to `bmassing@cs.trinity.edu`. Send program source as attachments. You can turn in your plots and input data as hardcopy or by e-mail; I have a slight preference for e-mail and a definite preference for something easily readable on one of our Linux machines — so, PDF or PNG or the like (in the past I think some students have sent me Excel spreadsheets, which — I'd rather you didn't). Please use a subject line that mentions the course number and the assignment (e.g., "csci 3366 homework 4").

5 Honor Code Statement

Include the Honor Code pledge or just the word "pledged", plus *at least one of the following* about collaboration and help (as many as apply).[1] Text *in italics* is explanatory or something for you to fill in. For programming assignments, this should go in the body of the e-mail or in a plain-text file `honor-code.txt` (no word-processor files please).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*

[1] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

6 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what about the assignment you found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).