IMPLEMENT AND EVALUATE SEQUENTIAL PROGRAM

Homework 1

By

[redacted]

CS 732 Parallel Computing

January 23, 2017

Problem Specification

Design and implement a sequential “Game of Life” program. This program involves creating a two-dimensional data structure to hold NxN cells that are set to either “alive” or “dead,” which will be referred to as the board. The size of N is set at run time from a command-line argument. The board is initialized by randomly setting the cells to either “alive" or “dead,” and the cells on the board are updated based on a specific set of rules. Updating does not modify the original board. Instead the computed value of each cell for the subsequent, updated state are stored in an auxiliary board, which then becomes the original board in the next generation. In other words, both the previous and current generations are stored in the board and auxiliary board, respectively, for each generation.

The rules for updating the board are as follows:

i. If a cell is alive in the previous generation and has 0 or 1 live neighbors, then the cells becomes dead in the current generation.
ii. If a cell is alive in the previous generation and has 2 or 3 live neighbors, then the cell remains alive in the current generation.
iii. If a cell is alive in the previous generation and has 4 or more live neighbors, then the cell becomes dead in the current generation.
iv. If a cell is dead in the previous generation and has exactly 3 live neighbors, then the cell becomes alive in the current generation.
v. If a cell is dead in the previous generation and has more or fewer than 3 neighbors, then the cell remains dead in the current generation.

The board will continue to be updated until the maximum number of generations is reached or until there is no change between two subsequent generations. The maximum number of generations will be specified by the user at run time using a command-line argument.

The Game of Life program must also be tested for correctness and have its performance measured.

Program Design

The problem description describes the algorithm to use: store data in a two-dimensional data structure, use a datatype that can have at least 2 different values, iterate over each value in the data structure to update the next generation by storing the new value in the updated board, and continue the process of updating until there is either no change in the board or the maximum number of generations is reached. Because the algorithm almost writes itself, the main important design decisions were choosing the datatype, the data structure, whether to put the Game of Life logic in a function or simply in main, compilation flags, and how to test for both accuracy and performance. Also, the programming language had to be selected.

C++ was used for this assignment because I have more experience with C++ than with C. Vectors were used to store the data because according to *The C++ Primer* (Lippman, Lajoie, Moo), arrays should be used only if the sizes of the arrays are known

at compile time. The sizes of the boards are taken as command line argument, so they cannot be known at compile time. This requirement for the program and the desire to write quality C++ code was the major motivation for using vectors instead of arrays to store the data required for the board.

Integers were used to represent the living and dead cells in the two-dimensional vector. Many data types could have worked for this, but using integers simplifies the program logic and reduces the lines of code. Technically only one bit is needed to represent alive or dead on the board, but using integers greatly simplifies the logic of checking the number of surrounding neighbors, because you can simply add them. Integers are larger than booleans and chars, but I was not concerned about running out of memory. 0 represents a dead cell, and 1 represents a live cell. The board is initialized with each cell being randomly assigned 0 or 1, however, the series will be the same at each execution because the seed for the random number generator isn't varied. This is preferable because it gives reproducible results.

Since the program is written for a single purpose, it seemed appropriate to place all of the Game of Life logic in main. One function exists for printing the board, which is in the submitted code for testing. Chrono time was used to measure execution time.

The assignment code is in the seq_gol.cc file and can be compiled using make or using the following command: g++ -o seq_gol -O3 seq_gol.cc -std=c++11. -O3 was used because the serial program runs very slow without extra compiler optimizations. During the development process, -Wextra and -Wall were used to display compiler warnings.

Testing Plan

There are two main types of testing that must be done for this assignment: correctness testing and performance testing. To test for correctness, I put the first official Game of Life logic into a new file called correctness_testing.cc and checked several specific boards. The correctness testing must check that the rules for the Game of Life are properly followed when updating the board for the next generation. To test for correctness, three boards were manually created and the Game of Life rules were run for exactly one generation. The boards were printed before and after running the one generation of Game of Life rules. The output was compared to a manual, pen-and-paper calculation of what the board should look like after one generation. The cases tested using these three board covers each possibility for relevant states of cells: dead cells with three neighbors should become alive, live cells with 0 or 1 neighbors should become dead, live cells with 2 or 3 neighbors remain alive, live cells with 4 or more neighbors become dead, and dead cells with more or fewer than 3 neighbors remains dead.

Performance was measured from the beginning to the end of the Game of Life logic loop, and the results were written to a file. For the submitted code, the output is not written to a file but printed to standard output instead. The initialization of the board was not included in the performance testing time because it's runtime is negligible compared to the Game of Life logic loop. Test cases for performance were chosen based on the assignment description, and each test was run three times to get an average run time.

Because the program is very simple and straightforward, there was not much else in terms of testing besides debugging during the development process.

Test Cases

Table 1 shows the results of the correctness testing, and Table 2 shows the results of the performance testing.

Table 1: Testing for Correctness

Test Case #	Initial board	Actual board after one generation	Expected board after one generation
C-1	100 001 001	000 010 000	000 010 000
C-2	000 110 011	000 111 111	000 111 111
C-3	111 011 011	101 000 011	101 000 011

Table 2: Testing Performance

Test Case #	Problem Size	Max Generations	Average Time Taken (Seconds)
P-1	1000x1000	1000	5.0207
P-2	5000x5000	1000	133.366
P-3	5000x5000	5000	603.668
P-4	10000x10000	1000	534.1517
P-5	10000x10000	10000	4665.4147

Analysis and Conclusions

Before beginning this section, here is the relevant information about the machine and compiler:

i. Machine name: grad-cuda.cis.uab.edu
ii. OS name and version: CentOS Linux release 7.2.1511 (lsb_release -a)
iii. Processor name: Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz (/proc/cpuinfo)
iv. Clock speed: 2806.118 MHz (/proc/cpuinfo)
v. Memory: 24514932 KiB (/proc/meminfo)

vi. Compiler name and version: g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-4)
vii. Compilation: g++ -o seq_gol -O3 -Wall -Wextra seq_gol.cc -std=c++11

The overall goal of implementing a Game of Life simulation was accomplished, and the accuracy of the program was established, as shown in Table 1. These goals were achieved by designing an algorithm that corresponded to the problem description and picking suitable datatypes and data structures for the problem. This was not a difficult problem, and programming the solution was very straightforward. Thus, there is not much to elaborate on regarding how implementing Game of Life was successful.

As expected, the sequential Game of Life program took a long time to run for large problems. An interesting aspect of the performance results is how predictably the runtime increases. Increasing the size of the board proportionally increases the runtime, as does increasing the maximum number of generations. In Table 2 it is clear that P-3 runs about 5 times longer than P-2, which makes sense because it's the same size problem running for 5 times as many generations. But P-2 runs about 25 times longer than P-1. This difference can be explained by realizing that that number of elements in a 5000x5000 matrix is not 5 times more than those in a 1000x1000 matrix. In fact, there are 25 times more elements in a 5000x5000 matrix than there are in a 1000x1000 matrix. All of this information shows that the algorithm's time complexity is O(n^2), and quadratic time complexity is less than ideal. Hopefully this can be improved by parallelizing the Game of Life loop.

If I were to do this program over again, it would probably be the same program in the end. However, I would like to experiment more with different possible datatypes for representing the Game of Life cells, such as char and boolean. As far as changing the design of the project, it may be able to be improved by putting the Game of Life logic in a function that takes a reference to the two-dimensional vector as its argument. Making such a function would not have a large impact on the runtime, but it could possibly make the code more readable and more generic. I would not suggest any additions to improve the working of the program because it is demonstrably correct. However, different compiler optimization, like maybe -Ofast, which I didn't use or experiment with, could have resulted in better performance.

References

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
C++ Primer, 5th Edition, by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo
http://www.learncpp.com/cpp-tutorial/713-command-line-arguments/
Dr. Pirkelbauer for C++ reference