CprE 308 Project 2: Multithreaded Game of Life

Fall 2012

The Project is due on **Wednesday, October 17th, at 11.59pm**, through Blackboard.
See the end of this document for handin instructions and the late policy.

1 Project

Your task is to write a multi-threaded program simulating John Conway's Game of Life. The game of life is an artificial process that is played on an $n \times n$ grid. Each cell in the grid is in one of two possible states, *alive* or *dead*. The initial states of all the cells are specified in advance, and the grid evolves from one generation to another according to the following set of rules. The state of a cell may change from generation i to generation $(i+1)$ depending on the state of the neighboring cells.The neighborhood of a cell consists of all cells that are to the left, right, up, down, top-right, top-left, bottom-right, and bottom-left. Note that some of these may not exist, for cells that are in the border of the grid. Thus, a cell may have as many as 8 neighbors, and as few as 3 neighbors.

Evolution of a Dead Cell: If a cell is dead in generation i, it becomes alive in generation $(i+1)$ if exactly three cells in its neighborhood are alive in generation i. Otherwise, the cell remains dead in generation $(i+1)$.

Evolution of a Cell that is Alive: Each cell that is alive in generation i evolves according to the following rules:

- If one or fewer cells in its neighborhood is alive in generation i, then the cell is dead in generation $(i+1)$, due to "loneliness".
- If four or more cells in its neighborhood are alive in generation i, then the cell is dead in generation $(i+1)$, due to "overcrowding".
- If two or three cells in its neighborhood are alive in generation i, then the cell survives and remains alive in generation $(i+1)$.

The game starts with an initial configuration, which we refer to as generation 0. The above rules are applied once to determine the state of each cell in generation 1. Applying

these rules again on the configuration in generation 1 leads to generation 2, and so on, and the game goes on for as many generations as desired. Also see wikipedia on the Game of Life and look at animations that are available on the web.

2 Requirements

Design and implement a multi-threaded program that simulates Conway's Game of Life as described above. Your program must be written using the "C" programming language and the "pthreads" threading package. The executable, called "life" takes in as many as five arguments:

```
life -n <grid size> -t <num threads> -r <num generations> -i <input>  -o <output>
```

- "grid size" is the size of the grid along each dimension, and must be an integer greater than 0. The two dimensional grid must have length and breadth both equal to this argument.
- "num threads" is the number of threads. This should be an integer greater than 0. This is the total number of threads used, including the thread that runs the "main" function. Thus, if "num threads" is equal to 5, then there are 4 more threads in addition to the main thread.
- "num generations" is the number of generations to simulate the game. This should be an integer greater than 0. The program is required to output the state of the grid at the end of this many generations.
- "input" is the name of the file containing the initial configuration. If this file is not specified using the "-i" parameter, then the program should assume that the input is from standard input (for example, use `scanf()` or equivalent). If the input file is unreadable, or does not have the right format, then the program should write out an appropriate error message, and exit.
- "output" is the name of the file to which the final configuration must be written into. If this file is not specified using the "-o" parameter, then the program should assume that the output should be written to standard output (for example, use `printf`). Again, the program should check for error conditions and take appropriate actions.

The input file containing the initial configuration will be in the following format. Suppose the grid is of size $n \times n$. Then, there will be n lines in the input file. Each line will contain the state of the cells in one row of the grid. A dead cell is represented by a 0 and a cell that is alive is represented by 1. Two bits are separated by white space, and the next row begins on a new line. An example of a 4 by 4 grid is as follows:

```
0 1 0 0
1 0 0 0
0 0 1 0
1 0 1 1
```

The output file must be in the same format as the input file. The program must first load data from the input file into memory to create the initial state of the grid. It should them use multiple threads to process the entire grid. After simulating the game for the specified number of generations, the output must be written to the output file. The program should be designed to avoid unwanted complexity.

An important issue is the synchronization between different threads. Say, there are 5 threads, and suppose that we assign one of the threads for the main function, and divide cells of the grid among the remaining 4 threads in some manner. Then, it is important to ensure that when a thread is trying to determine the state of a cell in the $(i+1)$th generation, the states of all its neighbors in the ith generation are available somewhere. This requires synchronization between the threads so that one thread does not get too far ahead of other threads who are operating on neighboring cells. You have to determine the synchronization required for achieving this.

3 Example

Suppose the program was run as follows:

```
life -n 4 -t 2 -r 1 -i input -o output
```

There should be a file named "input" with the following data:

```
0 1 0 0
1 0 0 0
0 0 1 0
1 0 1 1
```

The output file should be:

```
0 0 0 0
0 1 0 0
0 0 1 1
0 1 1 1
```

4 Speedup

Use the "time" command to measure the running time of your program with just one thread as follows:

```
time ./prog n 1 1 input
```

Next run it with a higher number of threads.

```
time ./prog n t 1 input
```

The speedup with t threads can be computed as the ratio between the time taken with t threads to the time taken with 1 thread. We will provide you with a set of example initial configurations, and for each such example, you are required to find the speedup of your program with 2, 4, 6, 8 threads and tabulate them.

5 Submission

Include the following in a tar.gz archive file and upload to Blackboard.

- A cohesive summary of what you learned in the project. This should be no more than two paragraphs.
- A description of the design on the program. Explain what are the responsibilities of different threads, and how you achieve synchronization between threads – explain the role of mutexes and condition variables, if any. Argue why this design is correct.
- Your source code; commented and formatted neatly. Use good programming practices – create methods to abstract large bodies of code and code common to multiple methods, only use global variables when you must, and check syscall or system dependent libcall return values for errors – don't assume the OS will always succeed in fulfilling your requests.
- A Makefile to compile your project.
- The results of your experiments on the speedup, tabulated in a text file.
- Include everything in a single archive and upload in Blackboard. Name your archive `life-<lastname>-<section>.tar.gz`.

6 Grading

Your grade will depend on the following:

- Correctness: does your multithreaded program generate the correct output configuration? Does it handle error cases correctly? Does it work with different numbers of threads? We will test it on a range of inputs different from the ones provided to you for experimentation.
- Does your code allow for parallel processing by different threads?

- How clear is the design, and have you explained your design decisions satisfactorily?
- What is the speedup achieved by your program on different inputs, with a given number of threads? Note that your speedup will also depend on factors other than your program, such as how many cores does your machine have, and which other processes are running simultaneously. Try to run your code on a machine with at least 8 cores (the lab machines fit the bill), and try to choose a time when there are no other users active on your machine.
- Clarity and readability of your code.

Late Policy You will lose 10% per day for one week. After one week, no credit will be given.