CONWAY'S "GAME OF LIFE"

Homework # 3

By

Marcelo Torres

CS 481/581 High Performance Computing
October 13, 2024

Problem Specification:

The objective of this assignment is to design and implement a simulation of Conway's "Game of Life," a cellular automaton devised by mathematician John Horton Conway. The Game of Life is played on a two-dimensional grid of cells, where each cell can be either alive or dead. The state of the cells evolves over discrete time steps according to a set of rules based on the states of neighboring cells. The rules are applied differently to living and dead cells. For living cells, if a cell has less than two living neighbors, it dies by underpopulation. If a cell has two or three live neighbors, it remains alive. Finally, if a live cell has more than three neighbors, it dies by overpopulation. Dead cells, on the other hand, remain dead unless they have exactly three live neighbors, in which case they become live cells by reproduction.

The goal of this project is to demonstrate the effect of multithreading on Conway's "Game of Life" by writing and running the program with larger and larger thread counts and constant iterations and field sizes. The program must be able to calculate the next game board and write results to an output file. The program must also be able to check results against the contents of an output file to verify that changing the number of threads does not affect the accuracy of the output. To that end, the program designed must use a dynamically allocated game board and accept the board size, maximum iteration count, number of threads, output directory, and test file as input.

Program Design:

In this iteration of my design, I reworked my project structure to achieve significantly faster results. All code can be found in the `homework3Submission` directory in the GitHub repository. All test results can be found in ` homework3Submission/test-results` in the GitHub repository. At first, I copied the design of the program given to use by the instructor, implementing the same logic and board design in my code. However, there still existed a significant gap in performance between my program and the provided solution. I realize that the only way to overcome this gap was to move my program elements into the same file. Therefore, I moved all program elements necessary to execute an iteration of the game of life into one file, GameOfLifeStandalone.cpp. The two included files, LibraryCode.hpp and FileIO.hpp contain utility functions used to set up the problem, and do not have significant impact on performance. After writing the standalone approach, my program achieved similar performance to the provided program. However, my goal was to achieve a faster runtime than the provided code. To this end, I redesigned the logic used to evaluate changes for cells. I started by attempting a window-based approach in which cells were read in vertical sets of three; however, this structure did not reduce runtime even though it reduced the number of reads from the array. Through testing, I realized that a significant portion of performance was lost because of branch conditions, so I redesigned the program to have minimal branches and minimal operations within each branch. The new logic loop is shown below:

```
591         int innerRowsNoUpdates = 0;
592         int innerColsNoUpdates = 0;
593         for (int row = groups.at(my_rank).first + border; row < groups.at(my_rank).second + border; row++) {
594             for (int column = border; column < columns + border; column++) {
595 
596                 int value = _arrays[offset][row - 1][column - 1] + _arrays[offset][row - 1][column] +
597                             _arrays[offset][row - 1][column + 1]
598                             + _arrays[offset][row][column - 1] + _arrays[offset][row][column + 1]
599                             + _arrays[offset][row + 1][column - 1] + _arrays[offset][row + 1][column] +
600                             _arrays[offset][row + 1][column + 1];
601 
602                 int oldVal = _arrays[offset][row][column];
603                 int newVal = (value == 3) ? 1 : (value == 2) ? oldVal : 0;
604 
605                 _arrays[nextOffset][row][column] = newVal;
606                 innerColsNoUpdates += (oldVal == newVal);
607 
608 //                    cout << "[" << row << ", " << column << ", s:" << sum << "] ";
609             }
610             innerRowsNoUpdates += (innerColsNoUpdates == columns);
611             innerColsNoUpdates = 0;
612 //              cout << endl;
613         }
614 
615         atomicRowsNoUpdates += innerRowsNoUpdates;
```

Figure 1: The logic to update cells in Conway's Game of Life

This loop stores the old value in an int then conditionally sets the newVal. The loop saves time by setting newVal to 0 by default and only changing it if there are 2 or 3 neighbors. By storing the new value in an integer, the program avoids having any costly branches, so the CPU does not waste compute guessing the wrong branch. Additionally, it accesses and stores the old value in every iteration, which is faster than conditionally accessing it. Lastly, changes are tracked by counting the number of times an update did not happen and storing them in a variable name innerColsNoUpdates. This variable is then checked to see if there were no updates for the entire row, and the result is added to a variable named innerRowsNoUpdates. The sum of the innerRowsNoUpdates variables from each thread is computed in atomicRowsNoUpdates. This system of tracking when changes do not happen is more efficient than a system that uses conditional statements, as it avoids the possibility of branching. The resulting code runs at about twice the speed of the given code for game boards of size 1000 with max iterations 1000.

After making these updates, I designed the multithreaded section. I used some simple optimizations to make this section efficient. First, I calculate the row groups only once at the beginning of the process. I split the board into groups of rows. If there are N threads, I create N groups. The overflow (size%N) is evenly distributed onto the first size%N groups. All groups are made of continuous sections of the board, and their first and last row indices are stored in an array for later use. The code to create these groups is seen below:

```cpp
std::vector<std::pair<int, int>> calculateRowGroups(const int rows, int numGroups) {

    if (numGroups < 0) {
        numGroups = 1;
    }

    if (numGroups > rows) {
        numGroups = rows;
    }

    int groupSize = rows/numGroups;
    int overhang = rows%numGroups;
    int previousOverhang = 0;
    int allocatedOverhang = 0;

    std::vector<std::pair<int, int>> rowGroups(numGroups);

    for (int i = 0; i < numGroups; i++) {
        rowGroups.at(i) = std::make_pair(i*groupSize + previousOverhang, (i+1)*groupSize + allocatedOverhang);

        if (i < overhang) {
            rowGroups.at(i).second += 1;
            previousOverhang = 1;
            allocatedOverhang++;
        } else {
            previousOverhang = 0;
        }
    }

    return rowGroups;
}
```

Figure 2: The code to compute row groups

For my multithreaded section itself, I call omp parallel outside my iteration loop. Then, I place a barrier at the end of each thread's iteration and wait until all threads have made their computations. I accumulate the rows without updates in an atomic variable for thread safety and efficiency. Then, in an omp single section, the offsets are increased, and the updates are checked. This structure ensures that each thread is created only once. Due to size constraints, a continuous photo of the loop could not be captured, but the inner logic can be seen in figure 1 and the outer loop logic can be seen below.

```cpp
    bool exit = false;
    atomic<int> atomicRowsNoUpdates = 0;

    #pragma omp parallel num_threads(numThreads) \
            default(none) \
            shared(_arrays, rows, columns, offset, nextOffset, groups, cout, iterations, exit, atomicRowsNoUpdates)

    for (int currentIteration = 0; currentIteration < iterations && !exit; currentIteration++) {

        int my_rank;

        #ifdef _OPENMP
        my_rank = omp_get_thread_num();
//          cout << "my rank: " << my_rank << endl;
        #else
        my_rank = 0;
        #endif
```

Figure 3: The outer loop for the main game computation

Testing Plan

The game logic was tested using small board sizes of 5x5 and a single iteration. The results were printed and compared to the website https://playgameoflife.com/ which provides a free interactive implementation of Conway's "Game of Life." An example is shown below:

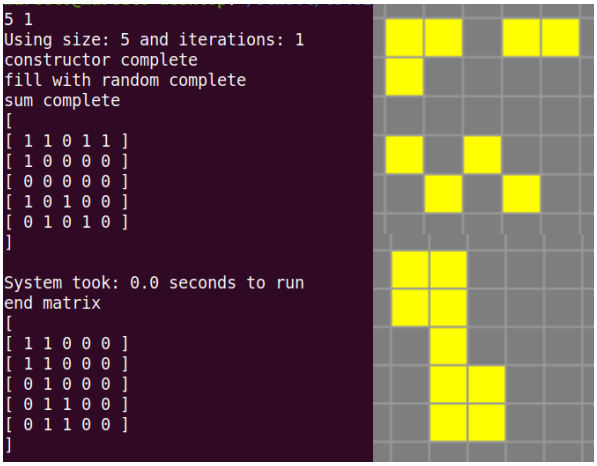

Figure 4: A validation of the game logic

The above figure depicts a printout of the game state before and after an iteration, as well as a graphic representation of the state taken from the website listed above. For this project, this testing was extrapolated to include the new code with different numbers of threads. The results of that testing were successful and can be viewed fully in `homework3Submission/test-results/testvalidityshSCRI.o115380` in the GitHub repository.

Validation for early stopping was tested using a 1x1 array of 1s and a 5x5 identity matrix. These tests were performed for different numbers of threads where possible, and the results were successful. The results of the first test with a 1x1 array and a single thread can be found in `homework3Submission/test-results/test_1_1.txt`. This test validates the ability of the loop to stop early on a single thread case. The results of the second test can be found in `homework3Submission/test-results/test_early_stop.txt`. This file contains the results of the diffs with the single thread version as well as the iterations performed on the board.

Test Cases

All Tests were run using a random, uniform distribution with a constant seed, which yields 50.0036% population, to populate the board. Tests were computed in sets of 3s on a 5000x5000 board with varying numbers of threads. The tests were all run using a release version of the code with the –03 compiler flag. The code was compiled using the ICPX and CMake libraries provided on the HPC and run using 28 cores on the asax015.asc.edu master node with 1500mb of memory. The results for each test were stored in an output file and compared to an output file generated using 1 thread prior to testing.

There were no early stops during the entire run. Additionally, there were no mismatches between the code's output and the test file. Results were identical for every run. The full result output can be found here: `homework3Submission/test-results/testhw3shSCRIPT-check-noPrint.o115379` in the GitHub repository. The second run using 16 threads took only 29.484 seconds, an outlier measurement. The reasons for this are unknown, but the run was replaced using a single run whose output can be found here: `homework3Submission/test-results/test5000500016shSC.o115435` in the GitHub repository. The results for each run are shown below:

Number of Threads	Run 1	Run 2	Run 3	Average Time (s)
1	340.232	337.517	330.104	335.951
2	167.558	168.230	172.236	169.341
4	87.907	87.857	87.827	87.864
8	46.195	46.788	46.263	46.415
10	39.739	41.912	37.717	39.789
16	49.511	29.484	48.716	42.570
20	40.351	43.423	39.260	41.011

Table 1: Runtimes for each test

Analysis

The results of the project were mostly as expected. Increasing the number of threads lead to increased performance up to a maximum, after which the overhead of the threads overtook their benefits. The speedup fell off at 10 threads, and the efficiency also dropped significantly after 10 threads. These results follow the expected behavior of a multithreaded program. At first, the program should gain performance; then, as threads are added, the overhead of multithreading should overtake the increased performance granted by the new threads. The plots of speedup and efficiency also reflect this expectation. A table of the raw values can be found below along with plots of the speedup and efficiency:

Number of Threads	Average Time (s)	Speedup	Efficiency
1	335.951	1.000	100%
2	169.341	1.984	99%
4	87.864	3.824	96%
8	46.415	7.238	90%
10	39.789	8.443	84%
16	44.866	7.488	47%
20	41.011	8.192	41%

Table 2: The speedup and efficiency for each number of threads

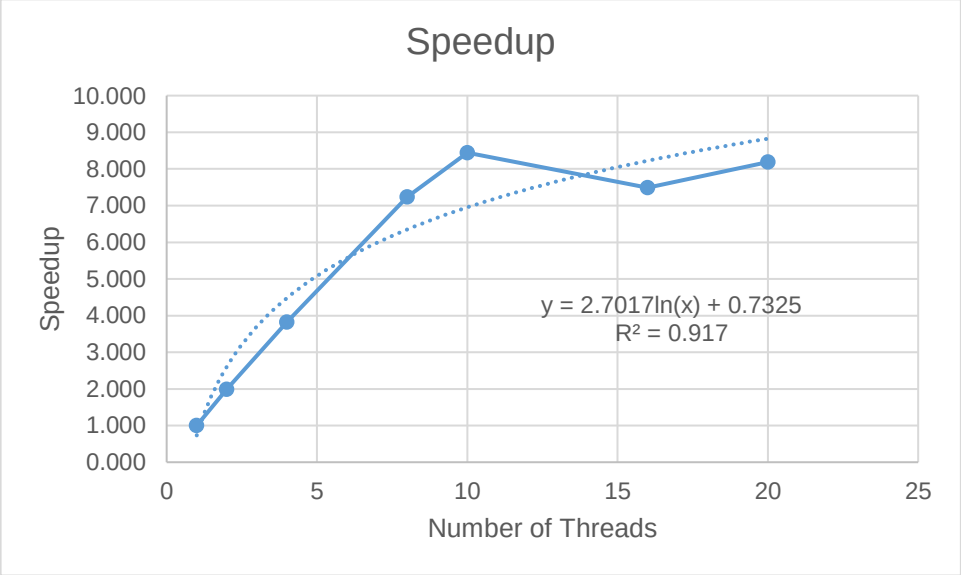

Figure 5: A plot of the speedup over different thread counts fitted with a logarithmic trendline

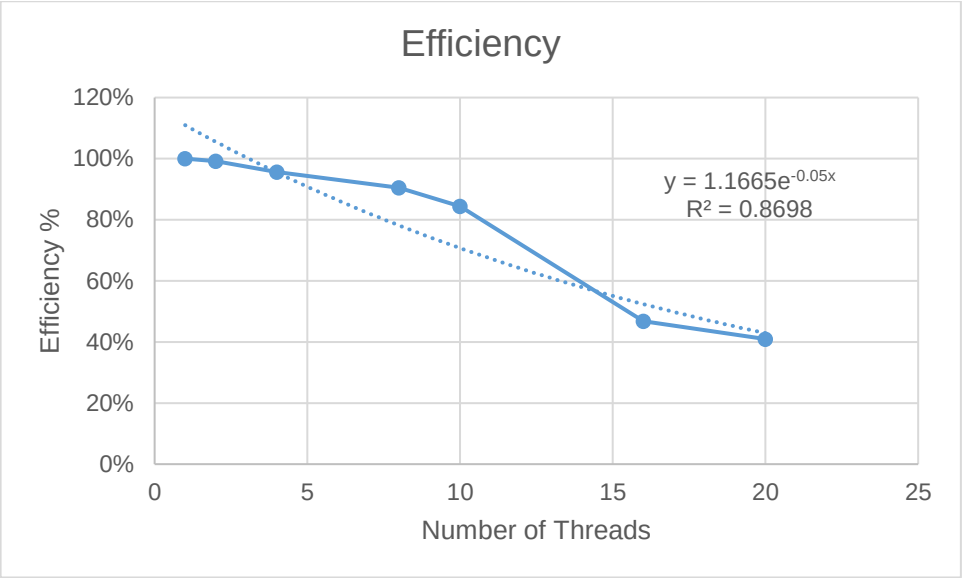

Figure 6: A plot of the efficiency of the program over different numbers of threads fitted with a linear trendline

As can be seen in the figures, the speedup and efficiency have an inversely proportional relationship, and both exhibit sharp behavioral changes after 10 threads. If more tests were conducted on higher thread counts, the speedup would probably level off and the efficiency would probably continue to drop while slowing down its rate of decrease.

Conclusions

My goal when beginning this iteration of the project was to create a system that could perform better than the code given to us by the teacher. By taking heavy inspiration from the given code and adding some ideas of my own, I was able to achieve this. Along the way, I learned some interesting lessons about what makes a program fast. First, I learned that the number of computations is not always as important as how those computations can be optimized by the compiler, the CPU, and the cache. The window-based approach I wrote had fewer reads from the array and less addition operations, theoretically making it faster than the basic example that accesses all 9 neighbors on each iteration. However, this algorithm was still slower than the basic example, which had fewer supporting operations and a much simpler structure. Additionally, the improvements that I made to the logic to iterate a cell's state were achieved by simplifying the code within the branches of the if statement and reducing the differences between each loop. These changes, although they introduced more possible computations and array accesses, were faster because they leveraged the cache's and CPU's abilities to optimize programs. The use of an efficient loop as well as the addition of OMP has made this project very efficient at calculating new game boards. I am proud of the work I put into this project and satisfied with the results.

If I were to try to improve this system further, I would explore ways to track cells that do not need to be changed. I would also investigate the window-based approach further. Although I was not able to make it efficient in this iteration of the project, I believe that the reduction in operations is valuable if implemented correctly. Looking back, I wish I had started with a simple system and worked my way up. Beginning the entire project with a multithreaded, class-based system took a long time and led to the creation of lots of inefficiencies. In the future, for efficient code, I will focus on developing and optimizing the simple stage first, then take those optimizations to more complex structures. Overall, I have learned a lot from this project and I am proud of the work I produced.

Sources

GitHub repo: https://github.com/Player1ce/CS481-HPC

Homework1-Report.docx by Marcelo Torres (URL not available)

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

https://playgameoflife.com