Uppsala University

High Performance Programming

1TD062

Individual Project

Jiayi Yang

March 22, 2019

1 Introduction

The *Game of Life* is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The universe of the *Game of Life* is an infinite, two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight *neighbours*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed; births and deaths occur simultaneously. The rules continue to be applied repeatedly to create further generations.

2 Problem Setting

In this project, we will implement a code that calculates the evolution of a cell system in a square field, according to the rules above, as an application of the *Game of Life*. The size of the square field and the time steps to be calculated are given by users as arguments. Also, users could control if the programme shows the pattern of the cell field, and how many threads will be used in the calculation. The initial pattern will be created randomly.

3 Solution

3.1 Algorithm

In the programme to simulate the cell system, at first, a $N \times N$ matrix p is created by dynamic memory allocation. Each element in this multidimensional array is set to be *0*, which means dead cell, or *1*, which means alive, to present the initial pattern of a cell system. This multidimensional array also contains the current status of all the cells in each timestep. In additional, another $N \times N$ matrix np is created by the same method, to contain the pattern of the next generation in each timestep.

In each timestep, for every cell in this system, the number of alive cell in its neighbours is counted and stored as s. Then for alive cells, if its s is 2 or 3, it will be alive on next step, otherwise it will die, while for dead cells, if its s is equal to 3, it will become alive, otherwise it will still be dead. The information about alive or dead is stored in matrix np. Since the system is bounded, which means not every cell has 8 neighbours, some conditions are added before counting s to judge if a cell is on boundary. After the calculation for every cell by loops, update the values in *p* to those in *np*, which means this timestep ends and *p* presents the new generation. This process is written in a function as *update*, which is called in every timestep.

Besides that, there is another function called *print* to output the pattern of given field, which is used to show the initial and the final pattern. After calculation, free the dynamic memories.

3.2 Optimization

The optimization attempts I tried are listed as below:

(1) Complier optimizations flags as *-O1*, *-O2*, *-O3* and *-Ofast*.
(2) Loop unrolling by using complier flag *-funroll-loops.*
(3) Use *const* keywords for *N*, *nsteps* and *nthreads.*
(4) Modify bounds checking:
    Instead of using conditions about rows and column for every cells to determine if it is on bounds, a set of row-controlled condition is used, which means, first determine if this cell is on the first or the last row, if so, don't count its neighbours on the row above or below it. Then determine the column. Here nested if-statements replaces serial if-statements.

3.3 Parallelization

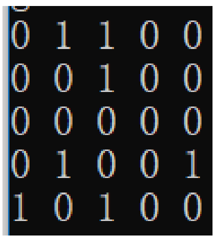
After analyzing the code with *gprof*, it shows that most time is spent in the function *update*. Therefore, *OpenMP* is used to parallelize this part, by parallelization on loops of rows with *dynamic* schedule.

4 Experiments

Implements in this section are with *AMD Ryzen 5 PRO 2500U w @ 2.0 GHz, Windows Subsystem for Linux, Ubuntu 18.04.1 LTS, gcc 7.3.0*

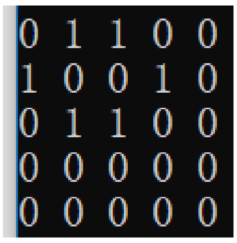
4.1 Algorithm

Calculating a 5 × 5 system, as Figure 1, for 5 timesteps, get a result as Figure 2, which is same as the result calculated by hand. After many times of implementing, the correctness of this code is proven.



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

Figure 1



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

Figure 2

Debugging with *valgrind*, it shows that there is no memory leak and no errors in the code.

4.2 Optimization

(1) Performance of different complier optimization flags:

flag	Time(s)
-O0	75.810
-O1	35.521
-O2	29.158
-O3	27.415
-Ofast	28.436
-O3 -funroll-loops	28.042

(N = 5000, nsteps = 100)

As a result, *-O3* is the best complier optimization flag. However, loop unrolling by complier does not improve performance, I guess the reason is, for each cell an integer *s* should be created and store some value in it, and with loop unrolling, two different integers would be created and store value, so it could not save time.

(2) Improvement with *const* keyword

	Time(s)
With *const* keyword	26.939
Without *const* keyword	27.415

(N = 5000, nsteps = 100)

The *const* keyword tells that this variable is never changed. Therefore, in this code, the size of the field *N* and the total number of timesteps *nsteps* are "constant" variable. Although there is not much improvement with *const* keyword, using *const* keyword in this code is also an efficient optimization method in theory.

4.3 Parallelization

(1) Parallelization for loops

There are two choices about how to parallelize the update function, parallelize the *i*-loop, which is a loop for rows, or parallelize the *j*-loop, which is the inner loop on elements. The performance of these two parallelization methods is shown as below:

	Time(s)
parallelize the *i*-loop	11.183
parallelize the *j*-loop	12.615

(N = 5000, nsteps = 100, with 4 threads)

The result shows that parallelization on outer loops is faster than parallelization on inner loops. The reason of that is parallelizing the inner loops will create a lot of overheads, which takes up time.

(2) Performance of different *schedule* types:

type	Time(s)
static	11.232
dynamic	10.825
guided	11.164

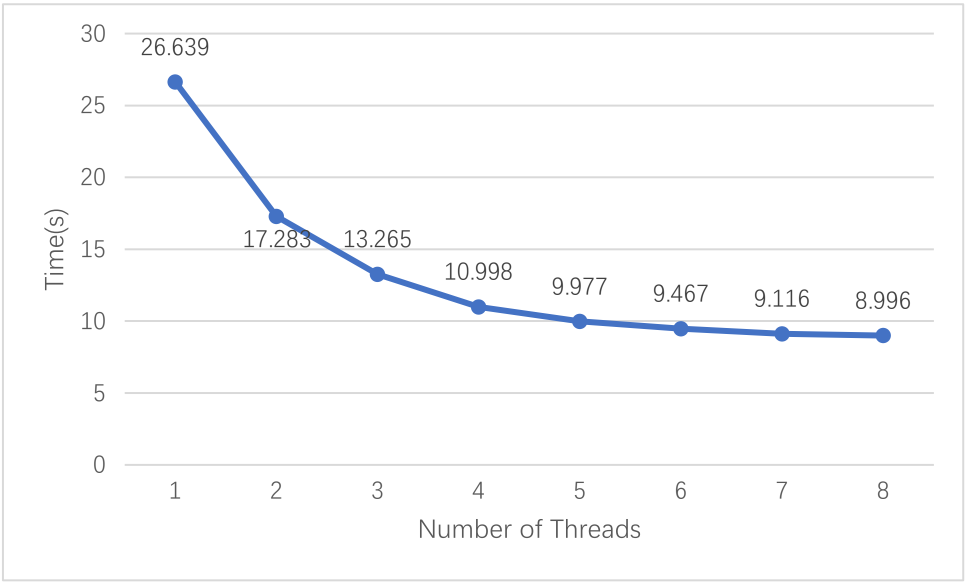
(N = 5000, nsteps = 100, with 4 threads)

Size of chunk	Time(s)
2	11.707
3	12.694
4	11.453
5	12.516
7	13.105
50	12.546
1250	13.538

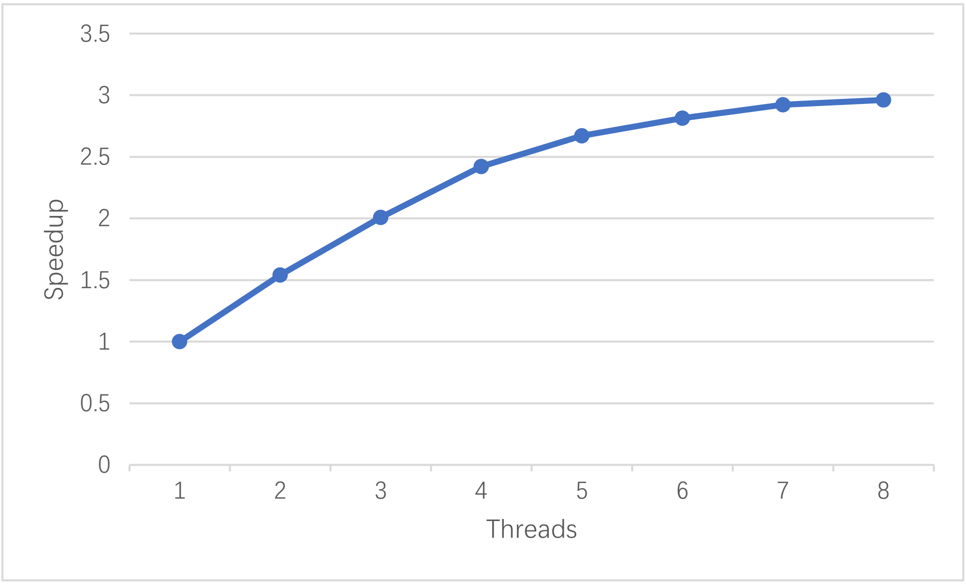
(N = 5000, nsteps = 100, with 4 threads)

As a result, dynamic scheduling is the fastest, because in this schedule, as soon as a thread is ready, it gets a new chunk. Because of the best size of chunk is related to *N* and *nthreads*, here leaves it as default, which is *N/nthreads*.

(3) Performance with different number of threads

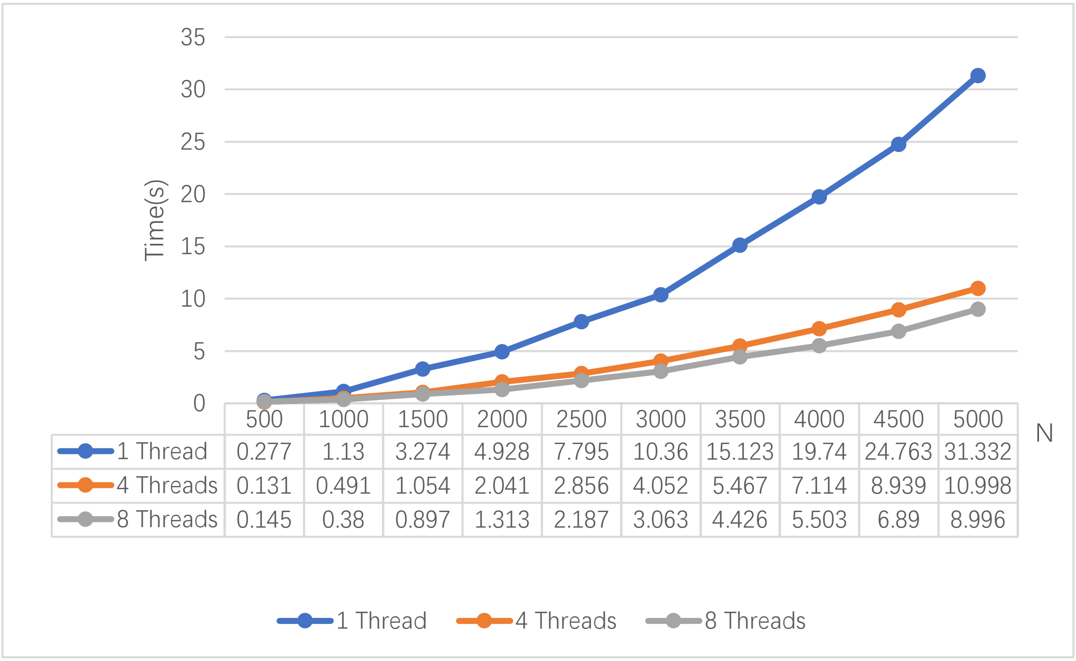


(N = 5000, nsteps = 100)



There is not much difference from 5 threads to 8 threads, and the speedups are smaller than expectation, it might because of overhead.

(4) Performance on different *N* with threads (nsteps = 100):



	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
1 Thread	0.277	1.13	3.274	4.928	7.795	10.36	15.123	19.74	24.763	31.332
4 Threads	0.131	0.491	1.054	2.041	2.856	4.052	5.467	7.114	8.939	10.998
8 Threads	0.145	0.38	0.897	1.313	2.187	3.063	4.426	5.503	6.89	8.996

Implement with 8 threads will be slower than 4 threads when *N* is small, the possible reason is 8 threads make a lot of overheads.

5 Conclusion

Most optimization and parallelization methods tried in this project are efficient. -O3 flag could make 2.76 times speedup, while 4 threads parallelization could make 2.42 times speedup.

In a conclusion, optimizations by compiler are most efficient, therefore, the complier optimization flags should be first considered in programming. Parallelization creates new threads, and the threads could calculate different parts separately and simultaneously, which might make great improvement.

However, some attempts did not make improvement as much as expectation. The possible reason of the problem with more threads and little cells might because of overheads.

As a result, for a system with N = 5000 and nsteps = 100, the final programme after optimization and parallelization is 8.43 times faster than the origin code.

6 Reference

[1] Wikipedia contributors. (2019, March 15). Conway's Game of Life. In *Wikipedia, The Free Encyclopedia*. Retrieved 21:05, March 22, 2019, from https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=887946452

[2] Agner F. (2018). *Optimizing software in C++, An optimization guide for Windows, Linux and Mac platforms.* Technical University of Denmark.