Program Assignment 2

Report

CSC 4760

Fall 2014

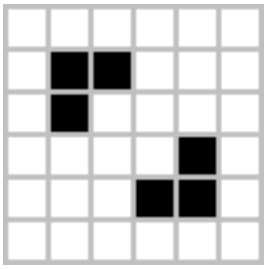
Parallel Programming

Game of Life

Author: Amilton de Camargo

Introduction

The purpose of this assignment was to write a program in C language where, given the problem size and number of threads, calculate the generations for the Game of Life and its remaining alive cells. An example is shown below.



Matrix *n* x *n* with some live cells.

An initial matrix, generated randomically, will set up the Game of Life to have its “zero” generation. All the next generations will rely on this first one.

The basic rules of this game are:

1. Matrices are called Grids, and they are considered infinite in size;
2. Any cell with less than two live neighbors dies, due to under-population;
3. Cells with 2 or 3 live neighbors lives on the next generation;
4. Cells with more than 3 live neighbors dies, due to overcrowding;
5. Any dead cell becomes alive if it has 3 live neighbors, due to reproduction.

The program written is capable of calculating any matrix size, as long as it fits into the memory. However, the bigger the matrix size, the longer it takes to calculate. As a rule for this program, the matrix is squared (same number of rows/columns), to make it easier to develop and run. Therefore, calling the program would look like: `$ ./game_omp [n] [g] [t]`

Where *n* is the matrix size (*n* x *n*), *g* is the number of generations to process, and *t* is the number of threads.

Any number greater than zero is allowed, although for this assignment, fixed numbers where set. Thus, it was possible to run the program and analyze its behavior while the matrix size and number of threads increases.

Objectives

- Write a program to perform the Game of Life in C language;
- Make it run multithreaded, using *pthread* and *OpenMP* libraries;
- Measure the time spent to calculate matrices of sizes: 10000, 5000, and 2000 for 100, 30, and 10 generations, respectively;
- Repeat the previous step, each time with one of the following number of threads: 2, 4, 8, 12, and 16.

Contents

Each generation will be stored in a matrix. As proved mathematically, an algorithm that performs a matrix access has a cost of n^2. It means that, given a matrix n x n, the number of steps to go through it is n^2. Also, the problem size occupied in memory is $2n^2$, for the current and the next generation. If considering the data type, it can be multiplied by its number of bits to know exactly how many megabytes (or gigabytes) are needed of RAM.

As described above, the purpose of this assignment was to measure the execution time when the program runs on different problem sizes, and also different number of threads.

Two methods to measure the efficiency of a parallel program are **Speedup** and **Efficiency**. Speedup shows how many times a program is faster when executed in parallel. Its formula is given by:

$$S = \frac{T_{serial}}{T_{parallel}}, \qquad time\ in\ seconds$$

If S is equals 1, it means that the program did not speed up. If the result is greater than 1, the program speeded up S times. If it is less than 1, the program slowed down.

Efficiency is a number from 0 to 1. It is a percentage of how the parallel program is taking advantage of its resources. Its formula is given by:

$$E = \frac{S}{P}, \qquad where\ S = Speedup\ and\ P = Number\ of\ threads$$

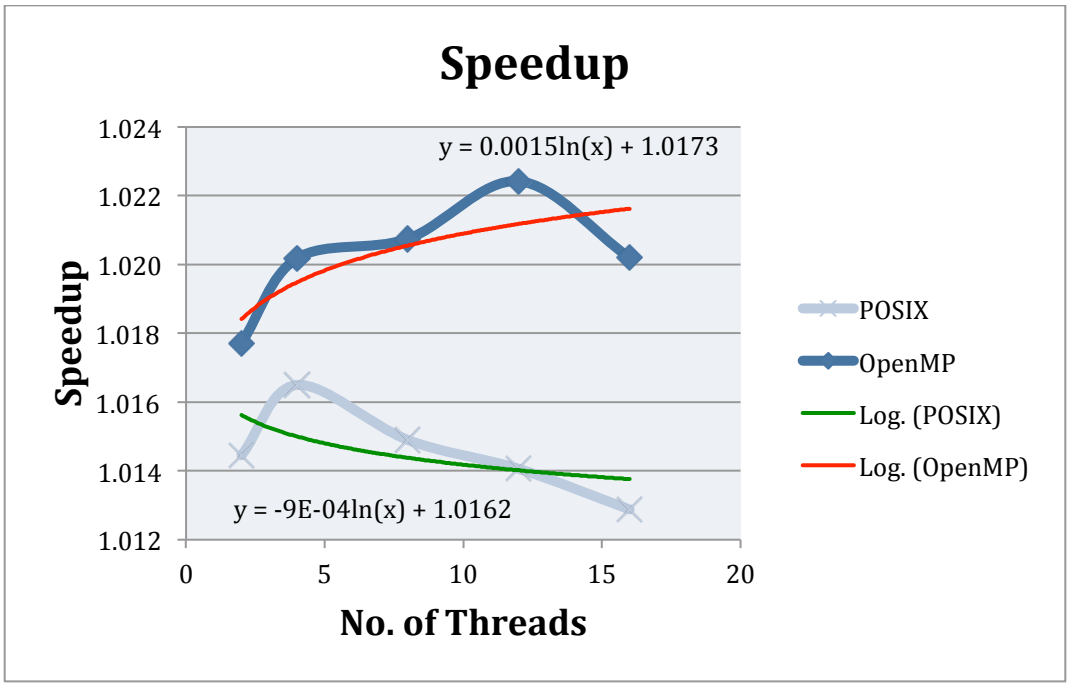
With these formulas, the time was measured and the respective values of Speedup and Efficiency were calculated. The results are shown below.

No. of Threads	Library	Matrix Size (n)	Generations	Tserial (s)	Tparallel (s)	Ttotal (s)	Speedup	Efficiency
1	POSIX Thread	10000	100	748.430	---	748.430	1.000	1.000
		5000	30	57.580	---	57.580	1.000	1.000
		2000	10	3.200	---	3.200	1.000	1.000
	OpenMP	10000	100	767.230	---	767.230	1.000	1.000
		5000	30	57.450	---	57.450	1.000	1.000
		2000	10	3.200	---	3.200	1.000	1.000
2	POSIX Thread	10000	100	2.770	742.580	745.350	1.008	0.504
		5000	30	0.710	57.020	57.730	1.010	0.505
		2000	10	0.110	3.120	3.230	1.026	0.513
	OpenMP	10000	100	2.660	749.930	752.590	1.023	0.512
		5000	30	0.690	57.770	58.460	0.994	0.497
		2000	10	0.110	3.090	3.200	1.036	0.518
4	POSIX Thread	10000	100	2.770	743.970	746.740	1.006	0.251
		5000	30	0.710	56.940	57.650	1.011	0.253
		2000	10	0.110	3.100	3.210	1.032	0.258

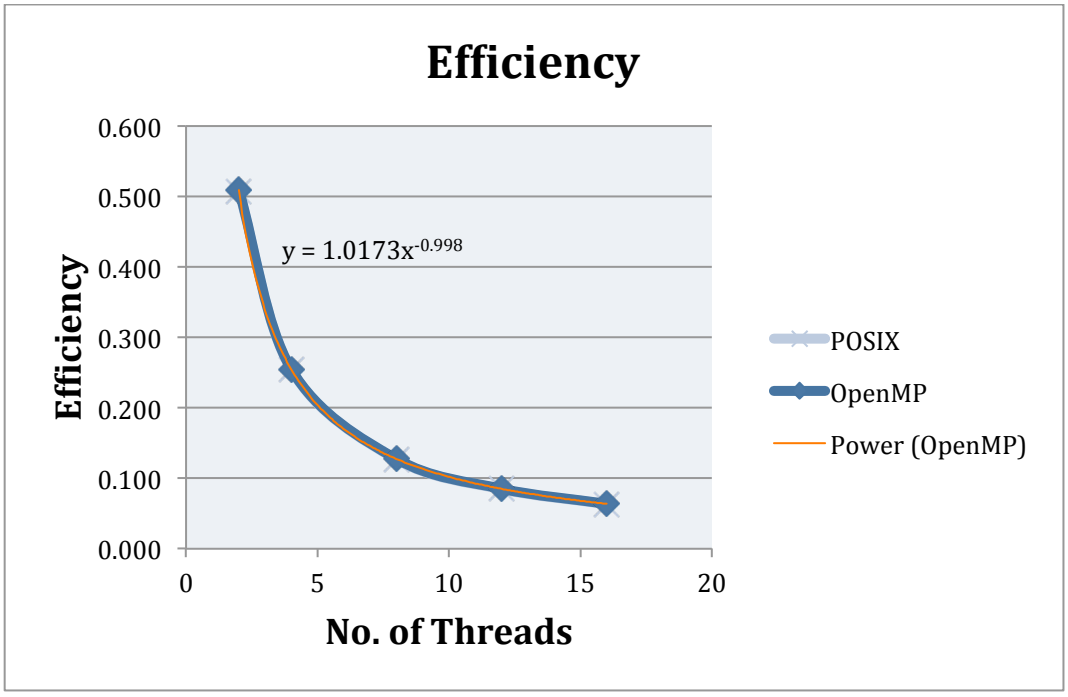
	OpenMP	10000	100	2.640	748.670	751.310	1.025	0.256
		5000	30	0.680	57.250	57.930	1.003	0.251
		2000	10	0.120	3.100	3.220	1.032	0.258
8	POSIX Thread	10000	100	2.910	745.720	748.630	1.004	0.125
		5000	30	0.710	56.890	57.600	1.012	0.127
		2000	10	0.110	3.110	3.220	1.029	0.129
	OpenMP	10000	100	2.760	748.970	751.730	1.024	0.128
		5000	30	0.670	57.320	57.990	1.002	0.125
		2000	10	0.130	3.090	3.220	1.036	0.129
12	POSIX Thread	10000	100	2.780	745.540	748.320	1.004	0.084
		5000	30	0.710	56.860	57.570	1.013	0.084
		2000	10	0.120	3.120	3.240	1.026	0.085
	OpenMP	10000	100	2.620	748.900	751.520	1.024	0.085
		5000	30	0.680	57.040	57.720	1.007	0.084
		2000	10	0.120	3.090	3.210	1.036	0.086
16	POSIX Thread	10000	100	2.750	745.870	748.620	1.003	0.063
		5000	30	0.700	56.850	57.550	1.013	0.063
		2000	10	0.120	3.130	3.250	1.022	0.064
	OpenMP	10000	100	2.880	751.730	754.610	1.021	0.064
		5000	30	0.690	57.200	57.890	1.004	0.063
		2000	10	0.110	3.090	3.200	1.036	0.065

Observation: All these tests were done in a laptop computer with a processor Intel Core i7 4th generation, 2.0GHz (up to 3.4 GHz with TurboBoost). It has 4 physical and 4 logical cores. Also, 8 GB of RAM DDR3 at 1666 MHz. Platform Apple MAC OS X Yosemite 64-bit.

In order to help seeing the behavior of the Efficiency and Speedup, graphs were plotted. They are shown below.



Graph 1: Speedup vs. No. of Threads



Graph 2: Efficiency vs. No. of Threads

Discussion

While running multithreaded, a program must synchronize all of them to fork its results. In this program, when using *pthread* library, the ***join*** function was used to do it. What it does is, given the thread handles, it waits until all of them return some value (or finishes). For the *OpenMP* version, it was not needed to explicitly write a barrier, because it is done automatically implicitly.

When looking to the Speedup graph, its behavior shows that it is higher at *OpenMP* library when the number of processors is 12. The maximum value for *pthread* is when it is running on 4 processors. Also, in this case, while the number of threads increases, the Speedup does not increase for *pthread*, making this algorithm be weakly scalable using this library, which means that the more processors are added, the less Speedup tends to increase. But while *pthread* decreases, *OpenMP* increases very slowly. However, at the beginning and the end, it is possible to see its value decreasing. It happens because everything in this library is managed dynamically, what has a cost for the computer to do so and it slows down.

The trending lines for Speedup shows that it behaves logarithmically, and it means that, after a certain point, the value will tend to be constant, where the program cannot be speed up anymore.

Now when looking to the Efficiency graph, as it should be because of the weakly scalability, the more the number of threads, the less the Efficiency. Note that it is a power function, which decreases its value quickly.

Comparing both threading libraries, it is possible to see that the Speedup is somehow different. While *pthread* tends to behave more smoothly, *OpenMP* goes on big highs and lows, which is expected, because it is not fully managed by the programmer. However, its trending line shows that it is a little bit more efficient than *pthread*, especially at the maximum numbers of processors. And, although *OpenMP* is slightly faster, its efficiency still exactly the same as *pthread*.

Conclusion

The Game of Life is not the easier problem to solve using an algorithm. It has some rules to be applied that can turn it complicated at the first time. Also, the bigger the matrix size, the longer it will take to finish calculating. To avoid this problem, parallel programming can help by dividing a huge matrix in small pieces and calculating each piece within a different thread/processor. But, in this case, there were no good results while parallelizing. Efficiency proved that the efforts to make it faster did not meet the expectations to have a very fast algorithm.

Speedup and Efficiency are very important methods to see if a parallel program is able to do its best effort. With this experiment, it was possible to put it on a practical way and see the results easier. It helps to conclude that this program is weakly scalable.

Both *pthread* and *OpenMP* are very useful libraries. However, *OpenMP* is the best one, because it can speed up the program a little bit more while keeping the same efficiency as *pthread*, but a lot easier to implement.

Not all problems will be speeded up efficiently with parallelism. Writing a parallel program is just a technique. After using it, it is necessary to determine how fast it can be and also how efficient it will perform its tasks. To do it, a good idea is to use the methods shown in this report. Graphically, it is better to see how the program behaves and determine whether or not it meets the expectations.