CSC 4760

Fall 2014

Program Assignment 3

Report

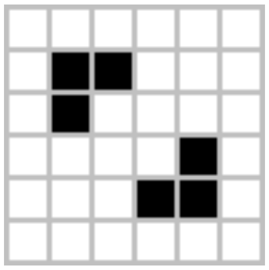
Parallel Programming

Game of Life in MPI

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 processes, 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:

```
$ mpi_run -np [p] ./game_mpi [n] [g]
```

Where *p* is the number of processes, *n* is the matrix size (*n* x *n*), and *g* is the number of generations to evaluate.

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, MPI library;
- Measure the time spent to calculate matrices of sizes: 10000 and 2000 for 100, 200, and 400 generations, respectively;
- Repeat the previous step, each time with one of the following number of processes: 2, 4, 8, 16, 32, and 48.

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.

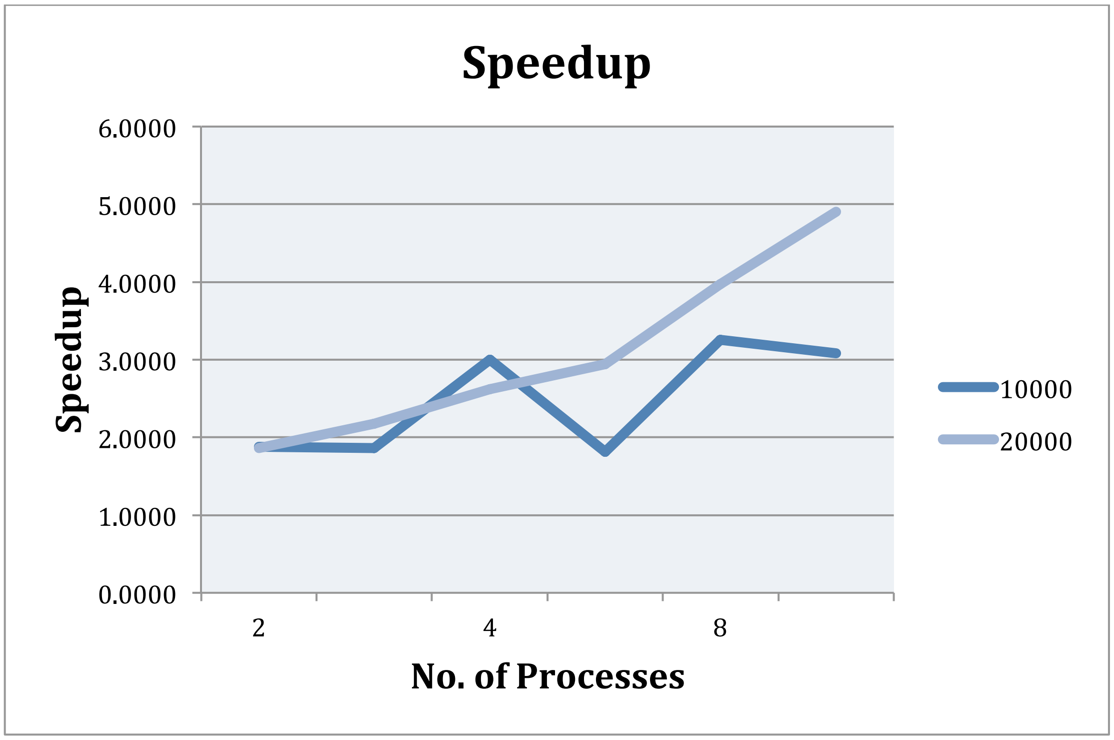
No. of Generations	Tserial (s)	Tparallel (s)	Ttotal (s)	Speedup	Efficiency
100	2.2303	223.6464	225.8767	1.7885	0.8943
200	1.8695	325.3852	327.2547	1.8440	0.9220
400	1.0102	401.2602	402.2704	1.9937	0.9969
100	7.9682	796.7756	804.7438	1.8826	0.9413
200	4.3683	1466.0344	1470.4027	1.9099	0.9550
400	4.1264	1682.0573	1686.1837	1.7835	0.8918
100	13.8681	661.6240	675.4921	0.6046	0.1511
200	8.9273	592.2935	601.2208	1.0130	0.2533
400	0.9372	201.1700	202.1072	3.9767	0.9942
100	41.4725	2196.4216	2237.8941	0.6829	0.1707
200	14.3649	900.9626	915.3275	3.1078	0.7769
400	8.0695	1096.4682	1104.5377	2.7361	0.6840
100	14.0934	361.3909	375.4843	1.1068	0.1384
200	8.6936	424.2418	432.9354	1.4143	0.1768
400	0.9228	123.5986	124.5214	6.4726	0.8091

100	42.2023	1152.1339	1194.3362	1.3019	0.1627
200	41.5434	1318.3296	1359.8730	2.1239	0.2655
400	7.1599	677.0517	684.2116	4.4310	0.5539
100	15.6225	195.0149	210.6374	2.0511	0.1282
200	10.0963	251.0552	261.1515	2.3899	0.1494
400	15.6803	798.8462	814.5265	1.0014	0.0626
100	37.6615	641.1109	678.7724	2.3397	0.1462
200	49.7855	1059.5304	1109.3159	2.6427	0.1652
400	7.0764	781.2642	788.3406	3.8399	0.2400
100	16.7918	114.5842	131.3760	3.4909	0.1091
200	12.5718	159.7149	172.2867	3.7567	0.1174
400	11.6548	318.8817	330.5365	2.5088	0.0784
100	53.7806	414.9956	468.7762	3.6145	0.1130
200	64.2365	896.7369	960.9734	3.1224	0.0976
400	22.1941	578.0072	600.2013	5.1902	0.1622
100	25.1515	115.4781	140.6296	3.4639	0.0722
200	17.5769	153.7835	171.3604	3.9016	0.0813
400	25.8876	425.3538	451.2414	1.8808	0.0392
100	93.2921	419.3957	512.6878	3.5766	0.0745
200	95.2840	778.9783	874.2623	3.5945	0.0749
400	23.1321	397.9954	421.1275	7.5378	0.1570

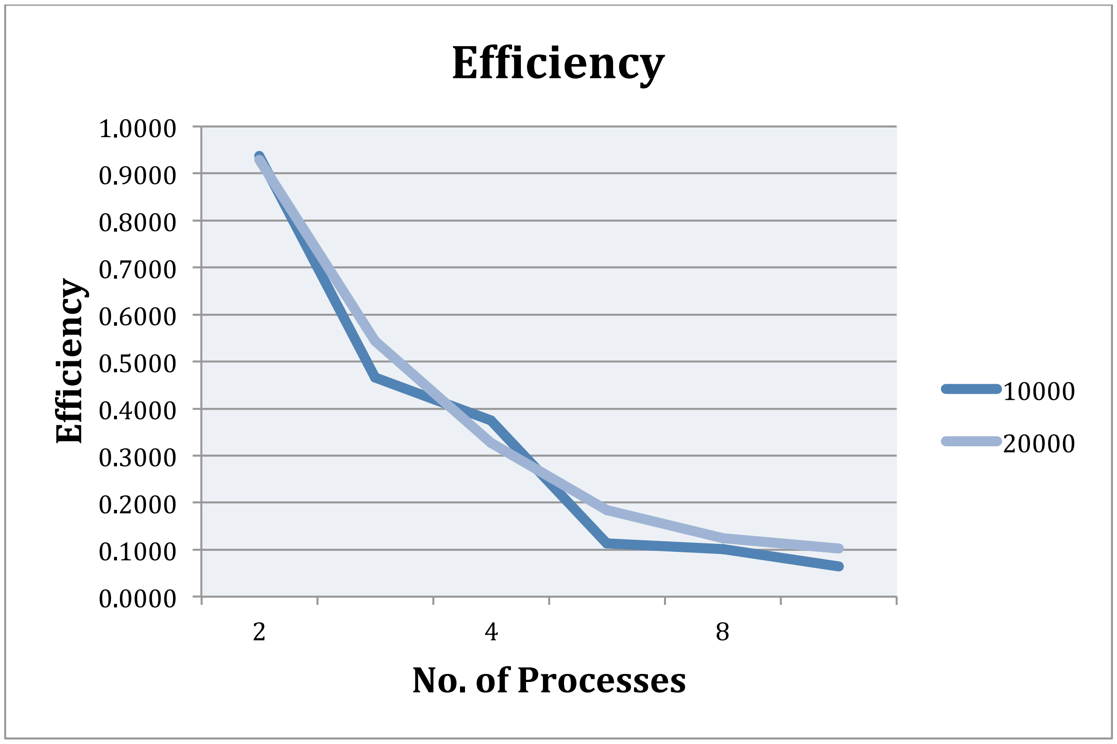
Observation: All these tests were done in the Computer Science's "Tiny" machine, using GPU1 through GPU4.

In order to get this calculation done, the entire matrix was partitioned among the processes. To do it, the algorithm selected a number of rows for each process to calculate. With that, each one had a work to do in parallel, and then returned its results for the main process (also called zero).

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



Graph 1: Speedup vs. No. of Processes



Graph 2: Efficiency vs. No. of Processes

Discussion

While running multithreaded, a program must synchronize all of them to fork its results. In this program, when using *MPI* library, the ***wait*** function was used to do it. What it does is, given the processes IDs, it waits until all of them return some value (or finishes) by messages.

When looking to the Speedup graph, its behavior shows that it is increasing. The more processes are added, the more Speedup tends to increase. The trending lines for Speedup shows that it behaves exponentially, and it means that, the more number of processes increases, the more the values will tend to be increasing, although every program cannot be speeded up after a top limit not reached on this experiment.

Now when looking to the Efficiency graph, as we can see, the more the number of processes, the less the Efficiency. Note that the value decreases quickly. It means that this algorithm is weakly scalable.

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 some good and some bad results while parallelizing. Efficiency proved that the efforts to make it faster did meet the expectations for a certain number of processes to have a very fast algorithm, while decreased on some instances.

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.

MPI is a very useful library. It is not the easier one to implement, but it can run under any platform, with reliability and warranty to work.

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.