Vrije University of Amsterdam

University of Amsterdam

Parallel Programming Practical

MPI Assignment Report

(Game of Life)

Ehsan Sharifi Esfahani

January 2017

Introduction

The Game of Life, also known simply as Life, is a classic example of a cellular automaton.[1] Every cell's next state is determined by its neighbors' states. The Game of Life is meant to show what will happen to organisms when they are put in close proximity to each other. Upon giving the Game initial conditions, each successive 'generation' (iteration) shows the evolution of the organisms. The Life have different applications. One interesting and surprising application of the Game of Life is that we can construct an initial pattern that will generate the prime numbers sequentially.[2] The goal of this assignment is to design and implement a parallel version of Game of Life for distributed memory using the given sequential code.

Game of life: Rules

Each 'generation' (iteration) of the game updates the current status of the location of living organisms. Each organism has eight neighbors as illustrated below.

1	*2*	*3*
4		*5*
6	*7*	*8*

A living organism continues to live if it has either two or three neighbors which are also living. An empty area brings to life a new organism if it has exactly three neighbors. An organism will disappear either with four or more neighbors, or with one or no neighbor.

Parallel algorithm

Given sequential algorithm is used to implement the parallel version. A master-slave methodology is used in parallel algorithm. Briefly, the whole grid breaks into parts by master and distribute them among slaves, each node communicates the boundaries between processes using MPI library.

A grid is two-dimensional because it consists of rows and columns, so the grid can be split into chunks of rows, chunks of columns, sub-grids, or various other configurations. In this algorithm, I choose to split the grid into chunks of rows.

Because, arrays is saved in C row by row, therefore, each matrix can save in a continuous memory space and it is easier to break them.

It is tried to give an equal number of rows to each node, except the last node which it might receive some more rows. In this program a variable named "last" is used to determine the number of rows assigned to last node. Therefore, MPI_Scatterv and MPI_Gatherv are used to scatter the jobs, a chunk of grid, and gather the result, respectively.

It is needed to keep at least two grids: the current grid and the next grid, determined with two variable called "new" and "old". Because it will not overwrite the cells in one grid before it have been referenced them all to determine the cells of the new grid.

Each node should send the first and last rows to its neighbors in each iteration. A function is declared, Snd_Rcv_Boundaries, to perform this action. The MPI calls used in this module send and receive messages using "non-blocking" methods to increase the performance. It means that the non-blocking methods allow the program to continue, regardless arrival or receiving the messages. But, program should be stopped by MPI_Waitall() when it is certainly needed to use the boundaries data.

OpenMP is an implementation of multithreading [2] that when OpenMP encounters a parallel section, it spawns threads to perform that section in parallel. If the section is enclosed in a loop, it dynamically assigns iterations of the loop to threads until all iterations of the loop have been executed. In this algorithm, rows are dealt with by looping over them, so we can leave the load balancing of threads to OpenMP. Multithreading for a specific loop is easily implemented using "#pragma omp parallel for private()" command in the program.

Performance analysis

Kernel part of each algorithm is a major part of execution time. In this algorithm, the major part is located in a function named doTimeStep(). This function is used to calculate the whole execution time. In a parallel program for making measurement on all CPUs individually, it is used the one CPU that takes the longest time to complete. Therefore, each node send its execution time to master and it finds the longest execution time.

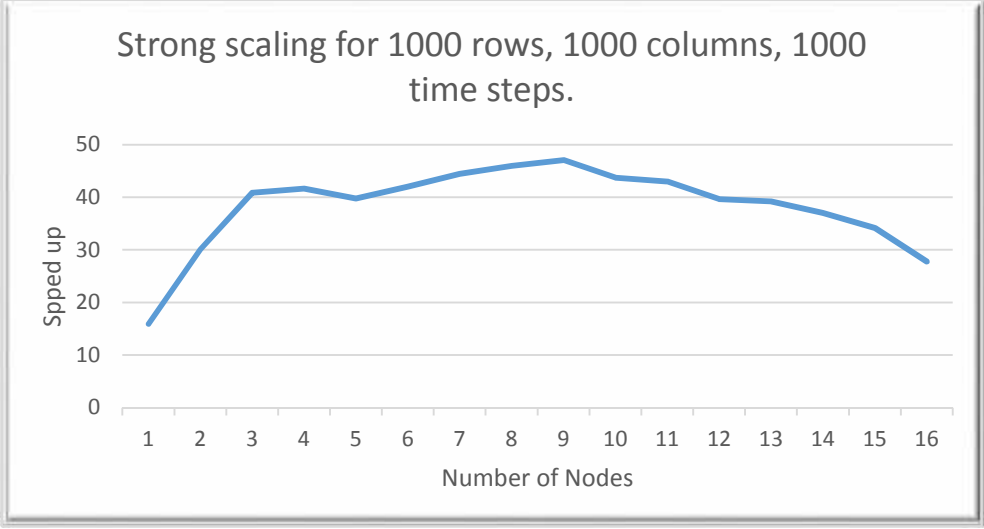
According to Amdahl's law, also known as strong scaling, the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system can be formulated as below:

$$S_{latency} = \frac{1}{f + \frac{(1-f)}{p}}$$

Where:

- $S_{latency}$ *is the theoretical speedup of the execution of the whole task;*
- *f is serial fraction of code;*
- *p is the number of processors.*

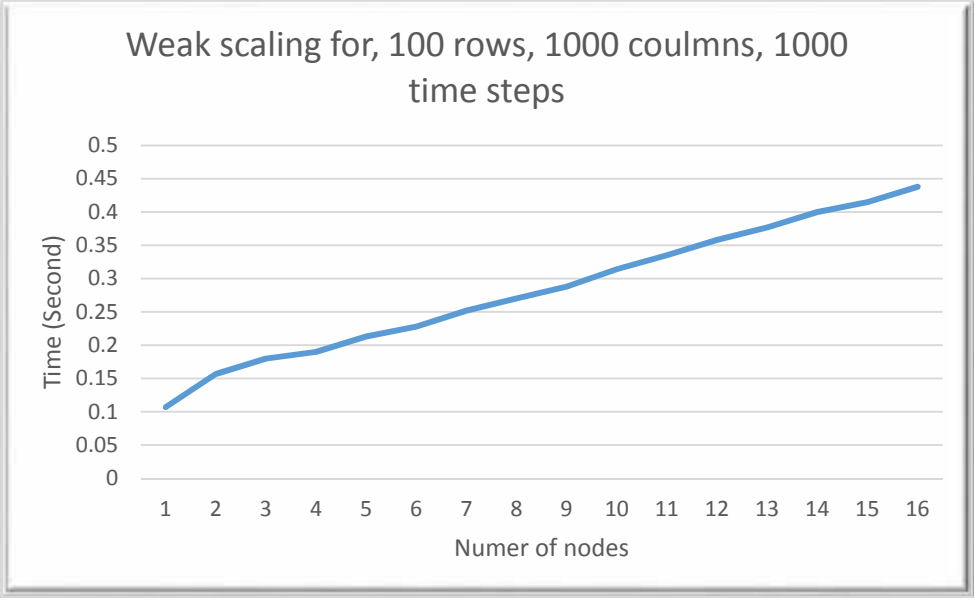
Therefore, Amdahl predicted that speedup will get to a maximum amount $(\frac{1}{f})$ *and it gets fixed when the number of processors get infinite. However, in practice, the situation is worse than predicted by Amdahl's Law due to: load balancing, scheduling, cost of communications and I/O. This matter is clear from the following chart obtained from the parallel algorithm of Game of Life. As it is depicted, speedup with one processor is about 16. This is due to the reason that multithreading, which is exploited using OpenMP, can be applied even with one processor, while at least two processor is needed to run a program in a distributed memory in parallel.*



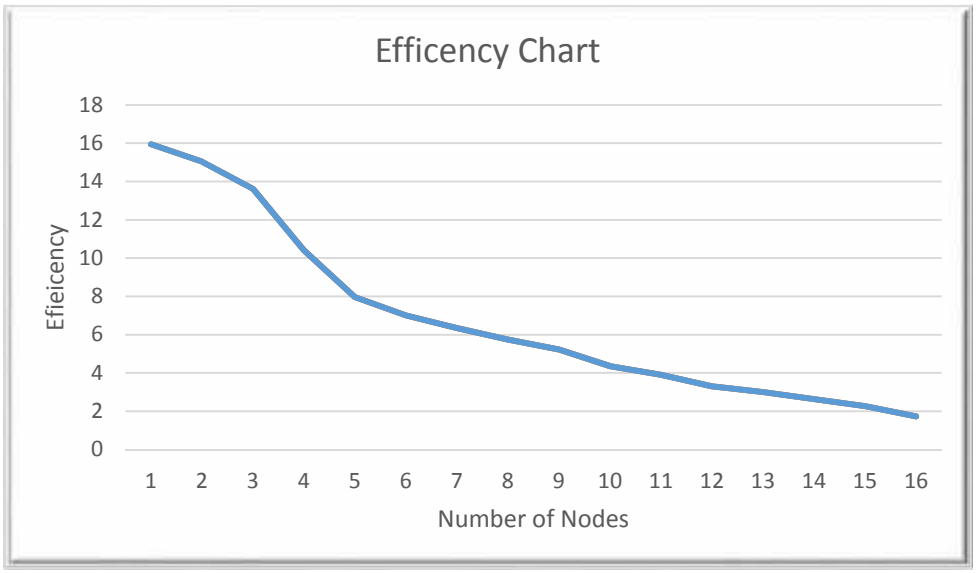
It is clear that speedup is increased, with a fixed problem size, to about 48 with 9 nodes. Then, it gradually went down when the number of processor is

increased. However, Amdahl's Law gives a rather pessimistic prediction regarding the speedup that we might hope to achieve. A key assumption in Amdahl's law is that the fraction of sequential work is fixed, regardless of the size of the problem. But, in many practical problems, that assumption is not true!

Moreover, the fraction of work that can be done in parallel often grows as the problem size increases; Gustafson's law takes this observation into consideration. Gustafson's law, known as weak scaling, rationalizes that as the size of the problem grows, the serial part will become a smaller and smaller percentage of the entire process. In other words, as programs get larger, having multiple processors will become more advantageous, and it will get close to n times the performance with n processors as the percentage of the serial part diminishes. It is shown in the chat below because the execution time and number of nodes with a fixed problem size per processor have a leaner relationship.



Efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized. As the chart below is illustrated, the efficiency is reduced when the number of processors is increased. However, It is possible to keep the efficiency in many parallel systems fixed by increasing both the size of the problem and the number of processing elements simultaneously, a system with this feature usually called scalable parallel system.



References

1- *Wikipedia Encyclopedia, https://en.wikipedia.org/wiki/Conway's_Game_of_Life, visited at Jan 2017.*
2- *Wikipedia Encyclopedia, https://en.wikipedia.org/wiki/Thread_(computing)#Multithreading, visited at Jan 2017.*
3- *Cellular Automata and Turing Universality in the Game of Life by Paul Rendell, presented by David Thue.*