Uppsala University

HIGH PERFORMANCE PROGRAMMING

Conway's Game of Life

Author:
Gabriel Evensen

August 20, 2021

Contents

1 Introduction

The project is focusing on the implementation of Conway's Game of Life. The project involves implementation of an algorithm, optimization of the code and parallelization with OpenMP. The report include a solution method which is describing the steps that were followed in order to get to the final solution. The Experiments section several three experiments are described and the results provided. The experimental part of the project has help with decision making for optimization as well as analysing the performance of the implementations.

2 Problem description

Conway's game consist of a universe of either dead or alive cells. The simulation takes place on a 2-dimensional grid. The setup of the cells within the grid are initially made by the user running the simulation. It is a zero player game which means that the evolution of the cells within the world is only depending on the initial state of the cells in the world requiring no further input. Every cell in the world has eight adjacent cells called neighbours.At every time step the following three rules are applied to each cell in the 2-dimensional grid:[3]

1. Any dead cell with three adjacent neighbours alive becomes a living cell.
2. Any living cell with two or three adjacent neighbours alive survives.
3. All other living cells dies. Similarly, all other dead cells stay dead.

3 Solution method

The solution method is based on the pseudo code presented in Algorithm 1.

Algorithm 1 Implementation of Conway's Game of life

world(i, j) ← Initialize a grid with dead (0) or alive (1 cells)

worldNextGeneration(i, j) ← Initialize a grid with the same size as *world*

loop for every cell in the initial grid *world*:

neighbours ← count the number of adjacent neighbours of the cell

if $world(i, j) = 0$ && $neighbours = 3$ **then**

$worldNextGeneration(i, j) \leftarrow 1$.

else if $world(i, j) = 1$ && $neighbours = 2$ || $neighbours = 3$ **then**

$worldNextGeneration(i, j) \leftarrow 1$.

else

$worldNextGeneration(i, j) \leftarrow 0$.

world ← *worldNextGeneration*. ▷ Copy *worldNextGeneration* into *world*

goto *Repeat the steps starting from the loop to simulate the evolution of cells within the world.*

The full and final solution method provided in code is available in section 6 Appendix. To get to the final solution method, several other alternative's for implementing the above provided algorithm were done and their performance was compared and evaluated.

3.1 Version 1: First simple implementation using the stack

In the first implementation of the algorithm the two worlds *world* and *worldNextGeneration* where initialized using the stack and the following lines of c code was written:

```
1 int world[row][col], worldNextGen[row][col];
```

The 2d-array of int was populated with dead (int 0) and alive (int 1) cells using the following loop:

```
1 for (i = 0; i < row; i++)
```

```
  {
    for (j = 0; j < col; j++)
    {
      world[i][j] = rand() % 2;
    }
  }
```

where every element in the 2d-array where randomly set to zero or one.

Every element in the 2d-array *world* was traversed and the number of adjacent neighbours of every cell in the 2d-array was calculated using the function *alive_neighbours*:

```
int alive_neighbours(int world[row][col], int m, int n)
{
  int i, j, count = 0;
  for (i = m - 1; i <= m + 1; i++)
  {
    for (j = n - 1; j <= n + 1; j++)
    {
      if ((i == m && j == n) || (i < 0 || j < 0) || (i >= row || j >=
   col))
      {
        continue;
      }
      else if (world[i][j] == 1)
      {
        count++;
      }
    }
  }
  return count;
}
```

where the function iterate over the eight adjacent neighbours and checks if the elements is alive. The first if-statement in the function controls that the index of the iterations since when traversing the eight neighbours of the elements on the border of the cell grid the array index can be out of bounds. It also avoid iterating over the cell in which the number of alive neighbours is counted for. After the number of alive neighbours of the cell has been counted the transitioning evolution of the cell is updated following the 3 rules stated in the above section 2. The updated state is stored in *worldNextGeneration*. Finally when all the element in *world* has been traversed and updated and stored, *worldNextGeneration* is copied

into *world* using the memcpy c function and the simulation continues of the Next generation of cells as long as the user of the program tells it to. See below:

```
for (i = 0; i < row; i++)
    {
      for (j = 0; j < col; j++)
      {
        // Count number of alive neighbours of cell
        count_alive_neighbours = alive_neighbours(world, i, j);

        // Game rules
        if (world[i][j] == 0 && count_alive_neighbours == 3)
        {
          worldNextGen[i][j] = 1;
        }
        else if (world[i][j] == 1 && count_alive_neighbours == 2 ||
   count_alive_neighbours == 3)
        {
          worldNextGen[i][j] = 1;
        }
        else
        {
          worldNextGen[i][j] = 0;
        }
      }
    }
```

3.2 Improvement Version 2: Allocate memory on the heap

The storage limitations using the stack to store the two worlds *world* and *worldNextGeneration* made it clear that the Algorithm has to implemented so that we allocate memory on the heap. The to world where now initialized using a 1d-array allocated with c function malloc:

```
int *world = (int *)malloc(row * col * sizeof(int *));
int *worldNextGen = (int *)malloc(row * col * sizeof(int *));
```

To improve good cache performance the data are stored and accessed row major order. The two dimensional grid that make up the world of cells are stored in row-major order which is a method for storing multidimensional arrays in linear storage. In order to do so the array *world* is initialized in a modified way using

$*(world + i * col + j)$ instead of $world[i][j]$ in the for-loop inserting dead or alive cells in the 1d-array *world*.

3.3 Improvement Version 3: Initialize the world with a boundary of zeros

In Figure 1 in subsection 4.1 it is clear that the majority of the running time is consumed in the function *alive˙neighbours* taking up 84% of the total running time of the program. In order to optimize the code the focus lies in optimizing this function which is done in what is now the third version of the code.

To improve the performance of the of the function *alive_neighbours*. The if statements in the alive_neighbours function is reduced into only containing one if statement which check if a neighbour cell is alive. This function now also avoid to iterate over the cell in which we counting neighbours for. The final function *alive_neighbours* is seen below:

```
int alive_neighbours(int *matrix, int m, int n)
{
  int i, j, count = 0;
  for (i = m - 1; i <= m + 1; i += 2)
  {
    for (j = n - 1; j <= n + 1; j++)
    {
      if (matrix[i * col + j] == 1)
      {
        count++;
      }
    }
  }
  i = m;
  for (j = n - 1; j <= n + 1; j += 2)
  {
    if (matrix[i * col + j] == 1)
    {
      count++;
    }
  }
  return count;
}
```

To make this reduction of if-statements possible the border of constant zeros surrounding the *world* is necessary. Instead of checking if the index is not out of bounds the elements on the border will all have eight adjacent neighbours. The for loop is divided in two, the first is checking the three cells above and below and the second for-loop will check adjacent neighbors to the right and left side of the cell.

3.4 Improvement Version 4: Parallelization

Majority of the time is consumed by the for-loop that is traversing the *world* array and counting neighbours of every cell and updating the *worldNextGeneration* following the rules of Conways's Game of Life. Since this for-loop is loop-independent makes it possible to parallelize. Using OpenMP which is an API for multi-platform shared memory multiprocessing programming the following line of code is added above the for-loop and The OpenMP functions are included in a header file called omp.h:

```
#pragma omp parallel num_threads(4)
{
    #pragma omp for schedule(static)
    for (i = 1; i < row - 1; i++)
    {
        for (j = 1; j < col - 1; j++)
        {
            count_alive_neighbours = alive_neighbours(world, i, j);

            if (world[i * col + j] == 0 && count_alive_neighbours == 3)
            {
              worldNextGen[i * col + j] = 1;
            }
            else if (world[i * col + j] == 1 && count_alive_neighbours
   == 2 || count_alive_neighbours == 3)
            {
              worldNextGen[i * col + j] = 1;
            }
            else
            {
              worldNextGen[i * col + j] = 0;
            }
        }
    }
}
```

where the clause Schedule is a specification of how iterations of associated loops are divided into a chunk and these chunks are distributed to the threads of the team. The number of threads in a parallel region is determined by the num_threads clause. If no chunk_size is specified, as in this case, the default value is to use as large chunk size as possible, giving approximately the same number of iterations to each thread.[2] Since the computer used has 2 cores and Hyper-Threading Technology enabled, the decision was made to run 4 threads in the parallel region.

4 Experiments

4.1 Experiment 1

In the first part and second part of this experiment the implementation and code considered is the one presented in subsection 3.2. The first aim of this experiment is to profile the code in version 2 to find out which routines and lines that consumes most time and where to put your effort in the optimization of the implementation.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 84.63    146.87   146.87 1677721600    87.54    87.54  alive_neighbours
 15.16    173.19    26.32                             main
  0.42    173.92     0.73                             frame_dummy

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
```

Figure 1: Results of running code version 2 with the profiler tool **gprof**.

In Figure 1 it is clear that the majority of the running time is consumed in the function *alive˙neighbours* taking up 84% of the total running time of the program. In order to optimize the code the focus lies in optimizing this function which is done in the version 3 of the code.

In the second part the aim is to run Valgrind which is a programming tool for memory debugging, memory leak detection. This is mainly done in order to assure that the code does not have any memory leaks but also to profile the storage efficiency in the code.

```
==7474== Memcheck, a memory error detector
==7474== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7474== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7474== Command: ./mainv8
==7474==
==7474==
==7474== HEAP SUMMARY:
==7474==     in use at exit: 0 bytes in 0 blocks
==7474==   total heap usage: 2 allocs, 2 frees, 4,194,304 bytes allocated
==7474==
==7474== All heap blocks were freed -- no leaks are possible
==7474==
```

Figure 2: Results of running code version 2 with Valgrind (*Valgrind ./a.out*).

As stated in the print of the valgrind tool the code has no memory leaks. In total there are 2 allocs and 2 frees.

4.2 Experiment 2

Different optimization flags are good for different kinds of code and loops. In this experiment different optimization flags will be tried for the code version 3. Version 3 is the final version of the code before parallelization. Every flag is compiled with version 3 of the code where the size of the world is 4096 rows and 4096 columns and there are 100 generations simulated. The version of the compiler is GCC Apple clang version 12.0.0. The simulations are executed on a computer with the specifications available in the appendix in the subsection computer specifications 6.1. The measured times are a mean value of five user times given from the `time` command put before executing file in terminal.

GCC Flag	Code version 2 (s)	Code version 3 (s)
`no flag`	91.148	59.420
`-O1`	38.410	18.917
`-O2`	17.280	9.825
`-O3`	13.092	9.178
`-Ofast`	13.087	9.170
`-O3 -march=native`	6.534	2.212
`-O3 -march=native -ffast-math`	6.525	2.216
`-O3 -march=native -ffast-math -funroll-loops`	6.523	2.221

Table 1: Performance of GCC Apple clang version 12.0.0 Optimization flags, the measured times are user times from the `time` command before executing file in terminal.

4.3 Experiment 3

In this third experiment the aim is to measure a potential speedup in the fourth version of the code when the program is parallelized using openMP. Every flag is compiled with version 3 of the code and the version 4 in which the code is parallelized, where the size of the world is 4096 rows and 4096 columns and there are 100 generations simulated. Since the GCC version Apple clang version 12.0.0 does not yet support the openMP API, the GCC version used in this experiment is GCC version 11.2.0 (Homebrew GCC 11.2.0). The simulations are executed on a computer with the specifications available in the appendix in the subsection computer specifications 6.1.

The measured times are a mean value of five user times given from the `time` command put before executing file in terminal.

GCC Flag	Code -v3 (s)	Code -v4 parallelized(s)
`-O1`	29.550	15.360
`-O2`	30.380	12.922
`-O3`	8.035	7.320
`-Ofast`	7.584	8.100
`-O3 -march=native`	7.684	6.650
`-O3 -march=native -ffast-math`	7.549	6.510
`-O3 -march=native -ffast-math -funroll-loops`	7.865	6.233

Table 2: Performance of GCC version 11.2.0 (Homebrew GCC 11.2.0) optimization flags, the measured times are user times from the `time` command before executing file in terminal.

5 Conclusions

The best performance received in this project was running the unparallelized code with the -O3 -march=native optimization flag with GCC Apple clang version 12.0.0 as compiler. The run time of a simulation with 100 generations and a world och 4096 (rows) times 4096 (columns) cells was 2.212 seconds with this optimization flag and compiler version. The speedup comparing with the same setup and version of the code and no compiler flag was almost 27, meaning that the run time was 27 times faster. Studying the results presented in Table 1. The most effective compiler optimization flag for boosting the performance of the algorithm was the flags -O3 and -march=native. Using those two flags together and the difference in run time of the algorithm is so small that it can be negligible.

After conducting Experiment 4.1, the results was showing that in order to boost the performance of the algorithm the focus must be to optimize the *alive˙neighbours* function which was consuming 84% of the total run time according to the gprof code profiler and the result presented in Experiment 4.1, in Figure 1. The changes made in the code version 2 made version 3 of the code to show a big speedup and it performance was boosted. Without any compiler flags used the speedup in version 3 was 1.53. For the fastest compiler flag setup, -O3 -march=native -ffast-math-funroll-loops for version 2 and -O3 -march=native for version 3, the speed up in the run time of version 3 of the code was close to 3. The

reason for the speedup and boosted performance in version 3 of the code may be because of the reduction of if-statement in the function of *alive˙neighbours*. Avoiding if-statements in loops if good for optimization since if-statements prohibits compiler optimization. Although, in version 3 of the code, the traversal of the cells when counting neighbours are in this version also traversing an outer boarder of zeros, meaning that $rows \cdot 2 + columns \cdot 2 + 4$ more cells are traversed when looking for alive neighbours. However, in this make it possible for reducing the if-statements in *alive˙neighbours* since the index bounds do not have to be checked at every traversed element.

The fastest run time running of the paralleized version 4 of the implementation was 6.233 seconds. This was slightly faster than the unparallelized version 3 which had a run time of 7.865 seconds. For lower parallelization level flags the speedup was between 1 and 2. The four threads running the computationally most demanding and time consuming part of the code and the function *alive_neighbours*, the overhead associated with setting up the parallel environment may have limited the potential speedup of the parallelization. Also the cache performance and memory access and storage efficiency may have been affected by the paralleization.

What could been further improved in this implementation is the cache-performance. Since the grid of cells is stored in a one dimensional array. The cells are organized and stored in row major order in memory. So, in order to boost cache-performance one could try to find a better solution for checking the 8 adjacent neighbours. Since the algorithm at now check three different rows when looking for adjacent neighbours, and the cells are stored in row major order, the eight neighbours are not store close to each other in memory. To find further speedup in this algorithm you need to look for iterating thru the entire cell grid. Instead of iteration through the whole grid each time step, one could keep a copy of all the cells that are changed which will narrow the work done in every generation simulated.Read more about this approach in chapter 17 and 18 in Michael Abrash's Graphics Programming Black Book, Special Edition. [1]

Instead of iterating through the entire cell grid each time, keep a copy of all the cells that you change.

This will narrow down the work you have to do on each iteration.

6 appendix

6.1 Computer specifications

Hardware Information	
Model Name:	MacBook Pro
Model Identifier:	MacBookPro14,1
Processor Name:	Intel Core i5
Processor Speed:	2,3 GHz
Number of Processors:	1
Total Number of Cores:	2
L2 Cache (per Core):	256 KB
L3 Cache:	4 MB
Hyper-Threading Technology:	Enabled
Memory:	8 GB
Boot ROM Version:	429.100.7.0.0
SMC Version (system):	2.43f11
Serial Number (system):	-
Hardware UUID:	498EFDB8-EC5D-57C4-8CDE-CF1E769DEADC8
`GCC` Version	GCC Apple clang version 12.0.0
`GCC-11` Version	GCC version 11.2.0 (Homebrew GCC 11.2.0)

References

[1] Michael Abrash'. *Graphics Programming Black Book, Special Edition.* URL: `http://www.jagregory.com/abrash-black-book/`. (accessed: 20.08.2021).

[2] Vivek Kale. *Loop Scheduling in OpenMP.* URL: `https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf`. (accessed: 20.08.2021).

[3] Wikipedia. *Conway's Game of Life — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Conway's%20Game%20of%20Life&oldid=1039084728`. [Online; accessed 20-August-2021]. 2021.