Laboratory Assignment: Game of Life parallelization and performance evaluation with OpenMP

Author: Giacomo Orsucci
E-mail address:
giacomo.orsucci@edu.unifi.it

Lecturer: Marco Bertini
Course:
Parallel Computing

Abstract

Nowadays, is more than ever necessary to parallelize code execution to fully exploit modern multi-core processors and to reduce significantly the execution time of computationally intensive tasks. For example, simulating complex cells interaction as in Game of Life, is computationally intensive computer graphics and artificial life application. The purpose of this assignment is to parallelize the Game of Life algorithm using OpenMP with C/C++ and evaluate its performance. The Game of Life algorithm simulates cellular automata with simple rules but is able to show complex, chaotic and self-organizing behavior. By leveraging parallel computing techniques, we aim to enhance the sequential simulation execution time. In order to achieve this, we'll be explored different versions of the algorithm, including a classic and basic parallel version and the benefits that can be obtained by introducing "cache tiling", SIMD instructions and code reorganization to fully exploit the hardware capabilities. This report details the implementation process, performance metrics, and analysis of the results obtained from the parallelized versions of the Game of Life algorithm with a complete but as compact as possible approach. After a brief introduction can be found a detailed and experimental oriented description of all the parallel implementations tested.

1. Introduction

1.1. Game of Life algorithm

The Game of Life, devised by mathematician John Conway in 1970, is a cellular automaton that simulates the evolution of a grid of cells based on a set of simple rules. It is defined as game (without interaction) because it involves a grid of cells that can be in one of two states: alive or dead. The state of each cell is determined by the states of its neighboring cells according to the following rules:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by over-population.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

These simple rules can lead to complex and emergent behavior, with patterns that can evolve, move, and interact in interesting ways. The Game of Life has been widely studied in the fields of mathematics, computer science, and artificial life, and has been used as a model for studying complex systems and emergent behavior.

More details about the Game of Life can be found at [1] and a possible implementation with rules exaplanation at [5].

1.2. Why parallelizing Game of Life?

1.2.1 Personal motivation

From a personal point of view, I decided to reimplement and parallelize the Game of Life algorithm as subject of this assignment because it has

been one of my first encounters with programming during high school. More specifically, it has been one of the first more complex and structured program I have ever written and I wanted to reengineer it using modern programming techniques and paradigms with the aim to parallelize it. A personal and old first implementation of the Game of Life algorithm can be found at [3].

1.3. Technical motivation

From a technical perspective, the Game of Life was chosen as a benchmark for parallelization due to its scalability challenges on large grids. While the algorithm has a computational complexity of $\mathcal{O}(N \times M)$ per generation ($\mathcal{O}(N^2)$ for quadratic grids), it fundamentally represents a stencil computation. In this pattern, updating a single cell requires reading the states of all eight neighbors. This results in low arithmetic intensity, meaning the processor spends more time moving data from memory than performing calculations. Consequently, the problem becomes memory-bound, making it an ideal candidate to analyze not just raw processing power, but also memory bandwidth efficiency and cache locality. In the following sections, we'll explore how code reorganization, in this context, is fundamental to improve code performance.

1.4. Test Machine specifications

The tests have been performed on my personal laptop with the following specifications:

- **CPU**: AMD Ryzen 7 3700U, 4 cores, 8 threads, base clock 2.3 GHz, max boost clock 4.0 GHz.
- **Cache**:
  - L1 Cache: 96 KB per core
  - L2 Cache: 512 KB per core
  - L3 Cache: 4 MB shared.
- **RAM**: 20 GB DDR4 (1 x 4 GB SODIMM DDR4 + 1 x 16 GB Row of Chips DDR4) at 2400 MHz.
- **Operating System**: Ubuntu 22.04 LTS.
- **Compiler**: GCC 11.4.0.
- **OpenMP version**: 5.2.

2. Experiments details

In this section are reported all the details about the different implementations, the meausurement methodology adopted and the results obtained.

2.1. Measurements details

- The measurements have been performed running the simulation for a certain number of grid updates, where an update is defined as the execution of *update_grid()*, that updates the grid following the Game of Life rules. In other words, according to the code provided in [4], an update is a complete iteration of the external loop in *update_grid()* or in another words the execution of the following pseudocode (just to give the general idea):

```
for each row:
   ...
   for each other column:

      count neighbors around cell (i, j);
      apply Game of Life rules to cell (i, j);
      ...
```

- Everything concerning the graphical part is not contemplated in the measurements, as the focus is on the performance of the *update_grid()* (Game of Life core) itself. The graphical part is used only to verify the correctness of the implementation and to perform indirect sanity checks.
- The duration time measures only the time taken to update a frame as defined above and is measured using `std::chrono::high_resolution_clock::now()`.
- Each experiment has been repeated 6 times, the first execution discarded and the average value obtained on the other 5 has been considered and used as the final measure.

- The number of updates performed to have a consistent execution time for each experiment has been fixed to 5000.

- The increasing quadratic sizes of grid are: 400, 800, 1000, 1200 because proven to be significant for our purposes thanks to some pretests not here reported because not relevant.

- The scheduling type normally has been chosen as static because the workload is uniformly distributed among the threads, but has been changed in some cases where a dynamic scheduling has been more appropriate.

- The number of threads used are: 1, 2, 4, 8, to cover all the possible configurations of the test machine. Tests with a number of threads greater the CPU threads would have introduced unnecessary context switching and overhead that would have distorted the measurements.

2.2. Implementations details

2.2.1 Graphics

The Game of Life graphical representation, fundamental to verify code correctness, is implemented using the SFML library [2] that is not parallelizable and so is not taken into account in experiments. Moreover, the graphical part is not relevant for the purpose of this assignment, having the only goal to verify the correctness of the implementation.

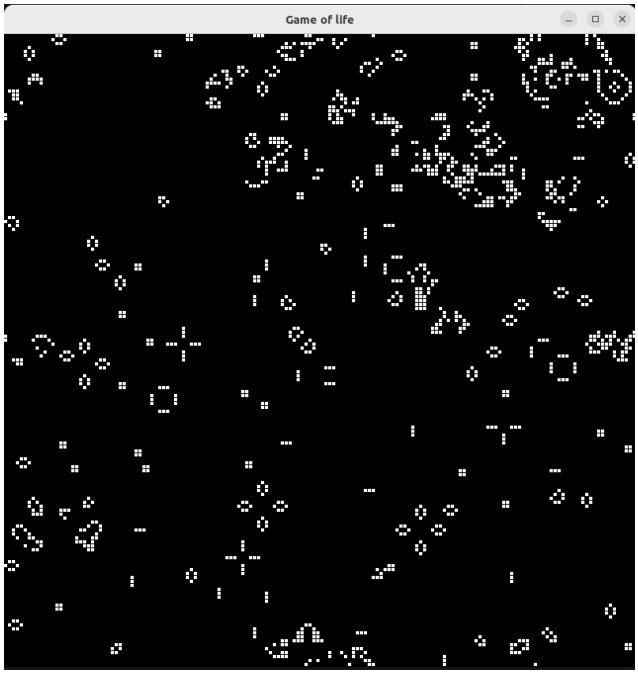

Figure 1. Example of graphical representation of Game of Life simulation with a 800x800 window.

2.2.2 Sanity checks and profiling

As mentioned before, the graphical part is used to verify visually the correctness of the implementation, but in addition some sanity checks have been implemented to ensure that the parallel implementations are not affected by race conditions or other issues that could compromise the correctness of the simulation. These sanity checks are included in debug mode (as visible in CMakeLists.txt):

```
-fsanitize=address,leak,undefined
```

where these flags, respectively, are:

- **address**: to detect memory errors such as buffer overflows, use-after-free, and memory leaks.

- **leak**: to detect memory leaks specifically, helping to identify memory that has been allocated but not properly deallocated.

- **undefined**: to detect undefined behavior in the code, such as null pointer dereferencing and integer overflows.

2.2.3 General optimizations

All the versions implemented share some general optimizations that have been applied to code to improve performance (in benchmark mode):

- **-O3**: compilation flag that tells the compiler to optimize as much as possible the code unrolling loops and inlining functions, where possible.
- **-march=native**: enables all the instruction sets supported by the host machine, allowing the compiler to generate optimized code for the specific architecture. Probably, the cost of this flag activation is the loss of compatibility with older processors that do not support the same instruction sets.
- **-mfma**: enables the use of Fused Multiply-Add instructions, which allows to perform multiplication and addition in a single instruction.
- **-ffast-math**: enables aggressive floating-point optimizations that may violate strict IEEE or ISO rules but can improve performance.
- **-fno-math-errno**: disables the setting of the global `errno` variable for floating-point math functions, which can improve performance translating, for example, *sqrt* in a single assembly instruction.
- **-mavx2**: enables the use of AVX2 instruction set, which provides advanced vector extensions for SIMD operations.

2.3. Code structure

In this section is reported in broad terms the code structure and the organization of the different implementations. The purpose is to make what follows and the repo at [4] more clear. The code is organized in a modular way, with different files for different implementations and functionalities. The main file is `main.cpp` that contains the common main and the graphical part of the code. *game.h* in *headers* folder contains the definition of *Config* struct that makes possible the code instrumentation to execute tests via *run_benchmark.py* and all the methods necessary to implement Game of Life and then implemented in *seq_game.cpp* for the sequential version, in *parallel_game.cpp* for the basic parallel version, in *tiled_game.cpp* for the parallel version with cache tiling and in *simd_game.cpp* for the parallel version with SIMD instructions.

2.3.1 Sequential implementation

The sequential implementation is the baseline for all the other implementations. This version is composed by the basic implementation of the Game of Life algorithm, without any parallelization or optimization in *update_grid()* and *count_neighbors()* methods (here and in the following the parameters will be ignored). This version is used as a reference to calculate the speedups achieved with the other parallel implementations, without any optimization or code reorganization. In other words, this is the plain version of Game of Life.

2.3.2 Parallel implementation

The first parallel implementation is a straightforward parallelization of the sequential version, with a first code reorganization to achieve better performance. In details, here we have the separation between update far the borders and update near borders (inside *update_grid()*) and so the use of 2 different methods to count neighbors:

- *count_neighbors_border()*
- *count_neighbors_noBorder()*

2.3.3 Tiled implementation

The idea behind this version is to improve cache locality by dividing the grid into smaller tiles and processing each tile independently. This can help to reduce cache misses and improve performance, especially for larger grids that do not fit entirely

in the cache. To translate in practice this intuition has been necessary to have, in *update_grid()*, a first loop to iterate over the tiles:

```
for (int ii = 0; ii < ROWS; ii += TILE_SIZE) {
   for (int jj = 0; jj < COLS; jj += TILE_SIZE) {

      // Update the cells
      ...
   }
}
```

also in this case we have a differentiation between the update of the borders and the update of the inner part of the tile, with the real optimization on the inner part of the grid.

2.3.4 SIMD implementation

SIMD (Single Instruction, Multiple Data) directives are also used in *tiled_game.cpp*, but really exploited in *simd_game.cpp* where the code is reorganized to be more suitable for SIMD instructions. This need has been clear after some pretests on the *tiled_game.cpp* version, where the performance improvement obtained with SIMD directives was significant, but not the best achievable. For this reason this version is organized in a way to be more suitable for SIMD instructions, with, for example, the use of precalculated indexes:

```
const unsigned char* row_up = &in_ptr[(i-1)*COLS];
const unsigned char* row_mid = &in_ptr[i*COLS];
const unsigned char* row_down = &in_ptr[(i+1)*COLS];
unsigned char* row_out = &out_ptr[i*COLS];
```

To make possible to write:

```
#pragma omp simd
for (int j = 1; j < COLS - 1; ++j) {
   int count = row_up[j-1]+ row_up[j]+row_up[j+1] +
               row_mid[j-1]+       row_mid[j+1] +
           row_down[j-1]+row_down[j]+row_down[j+1];

               ...
}
```

Moreover, here is used as much as possible branchless logic. This version works with a general *SCAN_SIZE*, but is particularly optimized for *SCAN_SIZE = 3*, that is the classic Game of Life case.

2.3.5 OpenMP directives

In *parallel_game.cpp*, *tiled_game.cpp* and *simd_game.cpp* the OpenMP directives are used to parallelize the code. They have in common the classic OpenMP directives:

- `#pragma omp parallel default(none) shared()`

  to create a parallel region where all the threads are spawned. The `default(none)` clause forces to explicitly specify the data sharing attributes of all variables used in the parallel region. This ensures a better control over the data sharing and helps to avoid unintended data sharing. The number of threads spawned is set previously in `main.cpp` with:

  ```
  omp_set_num_threads();
  ```

- `#pragma omp for schedule(static/dynamic)`

  to distribute the iterations of the for loop among the threads in a static manner when the load is certainly uniform and dinamically otherwise (as in *tiled_game.cpp*).

- `#pragma omp for collapse(2)`

  to collapse the 2 nested loops into a single loop to improve load balancing among threads. In our case is used to distribute the load associated with the tiles among the threads.

2.3.6 SIMD directives

SIMD instructions enables operations with expanded registers (256bit vs 32bit), this allows to perform the same operation on multiple data points simultaneously, for example 8 floats at a time with AVX2 instructions. Thus, it's immediately clear that SIMD instructions, if used properly, can significantly improve the speedup. With OpenMP, we can use:

```
#pragma omp simd
```

to instruct the compiler to vectorize the following loop using SIMD instructions. But as mentioned

before, all or much of the speedup achievable with SIMD can be lost if the loop contains branches (if-else statements). Thus, is profitable to implement branchless logic using arithmetic and bitwise operations to replace branches.

3. Experiments and results

In this section are reported all the plots obtained from the experiments performed and all the results are resumed in a compact way. Plots display the average speedup obtained with a different number of threads for an increasing grid size. All the experiments are executed for 5000 updates.

3.1. Parallel vs Sequential

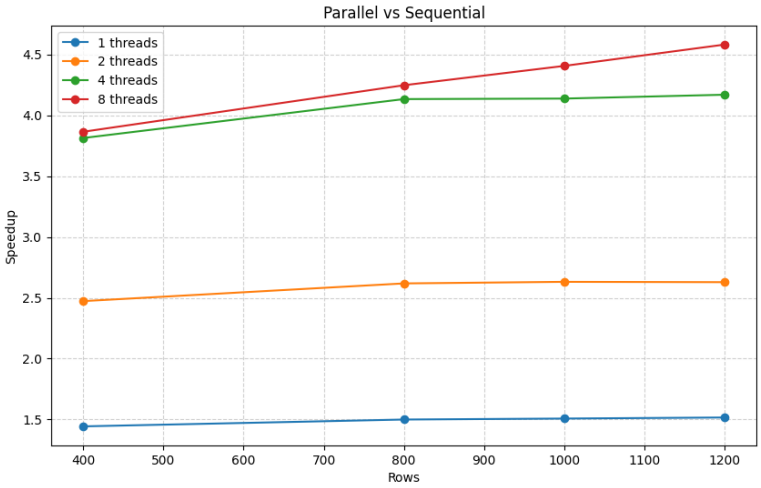

Figure 2. Speedups comparison of parallel version with the sequential one.

The experimental results demonstrate a significant speedup of the parallel implementation compared to the sequential baseline as the grid dimension and the number of threads increase. However, the performance curve exhibits a saturation effect typical of memory-bound applications.

Due to the low arithmetic intensity of the stencil operation (few calculations per byte fetched), the bottleneck shifts from computation to data transfer, leading to memory bandwidth saturation.

Notably, the transition from 4 to 8 threads (on a 4-core physical machine) does not yield a linear performance increase. This behavior is attributed to the limitations of Hyper-Threading in memory-intensive tasks. While Hyper-Threading allows two logical threads to share a physical core, they also share the L1/L2 cache and memory bandwidth. Since the cores are already stalling while waiting for data from the main memory, adding more threads increases resource contention rather than throughput.

Finally, it is worth noting that the parallel version achieves a speedup even when running with a single thread compared to the naive sequential version. This result validates the effectiveness of the code optimizations: by separating the border handling from the inner domain ('safe zone') and adopting a branchless logic, we reduced branch misprediction penalties and enabled the compiler to apply SIMD vectorization more effectively.

3.2. Tiled vs Sequential

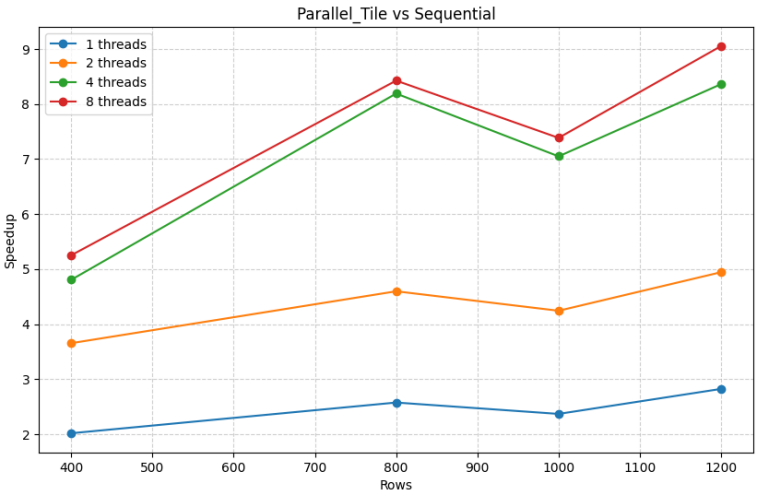

Figure 3. Speedups comparison of Tile parallel version with the sequential one.

As mentioned before, to further mitigate the memory bottleneck, a Tiling (or Blocking) strategy was adopted. The grid is decomposed into sub-blocks of size 128×128. Since each cell is an unsigned char (1 byte), a single tile occupies 16 KB, effectively utilizing 50% of the per-core L1 Data Cache (32 KB).

This sizing maximizes temporal and spatial locality, ensuring that the working set remains residency within the fastest memory tier during the

stencil computation. By minimizing cache evictions, this approach significantly reduces the pressure on the limited shared L3 cache and the saturated main memory bandwidth.

Consequently, the tiled implementation achieves superior speedups compared to the basic parallel version, particularly for large grid dimensions. However, a distinct performance drop is observed for the 1000×1000 grid size (especially with 4 and 8 threads). This anomaly is attributed to geometric inefficiency: the grid dimension is not aligned with the tile size, resulting in a high ratio of boundary tiles processed via the slower scalar path. Furthermore, the working set size for this specific resolution hovers around the capacity limit of the per-core L2 cache (512 KB), likely triggering cache thrashing

3.3. SIMD vs Sequential

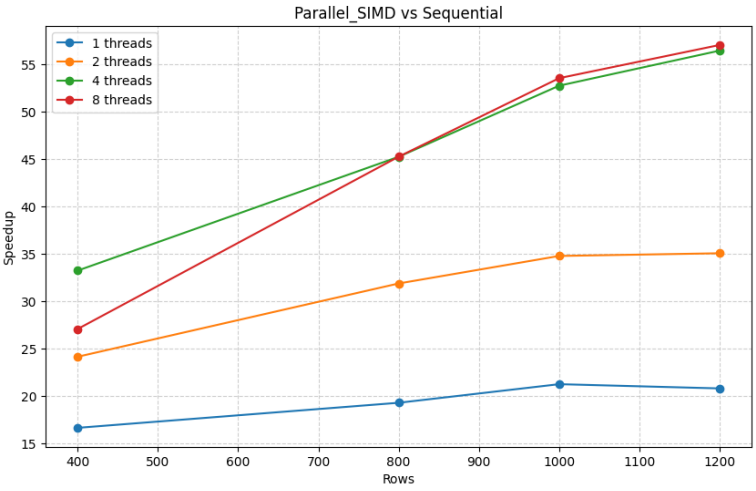

Figure 4. Speedups comparison between SIMD parallel version and the sequential one.

This iteration presents a code structure reengineered to maximize SIMD efficiency. By replacing complex index calculations with pointer arithmetic and adopting a branchless, unrolled logic, the implementation successfully exposes data parallelism to the compiler.

The performance improvement is substantial and honestly above expectations, highlighting the high cost of integer multiplication and branch misprediction in the previous versions.

Notably, the results regarding Hyper-Threading are particularly revealing in this context. The performance plateaus at 4 threads (matching the physical cores), with no gain observed when scaling to 8 logical threads. Consequently, enabling Hyper-Threading on multiple threads other than physical cores leads to resource contention rather than throughput increase, in this case resulting in the same performance as the 4-thread configuration.

4. Conclusion

In conclusion, in every case has been achieved a speedup with respect to the sequential version, demonstrating the effectiveness of parallelization and code optimization techniques. So, generally speaking, parallelizing the Game of Life algorithm with OpenMP has proven to be an effective way to improve performance, especially when combined with code optimizations such as cache tiling and SIMD instructions. Moreover, has been proven that Hyper-Threading is not always beneficial, especially in memory-bound applications like Game of Life. Eventually, we can conclude that the SIMD version is the best performing one, achieving a maximum speedup really above expectations that highlights very clearly how in Game of Life, more than data organization, is really fundamental to write code highly oriented in a branchless way, taking in account the cache hierarchy and the hardware capabilities to fully exploit the best code optimization possible.

References

[1] Game of life wikipedia. https://it.wikipedia.org/wiki/Gioco_della_vita.

[2] Sfml library. https://www.sfml-dev.org/.

[3] G. Orsucci. Game of life old code. https://github.com/Giacomo-Orsucci/Game-of-life-old., 2019.

[4] G. Orsucci. Game of life parallelization. https://github.com/Giacomo-Orsucci/Parallel_Computing_Lab, 2026.

[5] Yordanos. Game of life code. https://medium.com/@theyordanos/implementing-conways-game-of-life-in-c-e80f4448cd89.