HIGH-PERFORMACE OF CONWAY'S GAME OF LIFE,

Homework 2

By

Gowtham Prasad

CS 581 High Performance Computing
September 26. 2024

Problem Specification:

The objectives of this project are the following:

- utilize a version control system for software,
- submit jobs through a batch system, and
- evaluate a technical paper.

The goal of the C code is to create and run a simulation of Conway's Game of Life, a cellular automaton in which a two-dimensional array of cells evolves according to predefined rules. It was American mathematician John Von Neumann who first proposed the concept of a universal constructor, or a software that could replicate itself automatically by analyzing data, back in the 1940s, when life first originated. Von Neumann created a discrete game with a maximum of 29 distinct states for each cell in a two-dimensional grid of square cells to explore this problem. In this game, a cell's state in each generation is determined by the state of adjacent cells. This universal constructor fascinated British mathematician John Conway, who used it as motivation to study a new computable discrete world. Alan Turing's idea of a “universal computer,” or a device that can mimic any computational process by adhering to a brief set of instructions, served as the foundation for this study. Conway thus began to search for a computable “simple universe,” a condensed set of states and laws. After experimenting with numerous configurations on a two-dimensional grid to establish an optimal balance between extinction and unlimited cellular proliferation, he named this set of rules the Game of Life.

Each cell is categorized as either “alive” or “dead” based on the influences of its eight neighbors. The simulation must precisely update the grid over a number of generations and terminate when either a stable state is reached, or the maximum number of generations is reached. Only if a surviving cell in this generation has two or three neighbors will it survive until the next generation; otherwise, it dies from loneliness or overcrowding. Even a dead cell can spontaneously regenerate into a living one in the next generation if it has exactly three neighbors in the present generation. The seed of the system is the original pattern and is an N x N matrix. The aforementioned criteria are applied simultaneously to all of the seed's cells, whether they are living or dead, to generate the first generation. Births and deaths take place at the same time, and this discrete moment is frequently referred to as a tick. Every generation is solely dependent on the one before it. The guidelines are consistently followed in order to produce new generations.

Program Design:

To solve this problem, a two-dimensional array is used to represent the grid of cells, and an iterative approach is employed to simulate the transitions from one generation to the next.

For instance, two arrays are used to implement the grid: one to store the state of the current generation and another to record the state of the next generation. By using a method known as double buffering, which avoids needless memory allocation and grid copying, the software alternates between the two arrays after each generation. An additional layer of “ghost cells” is put around the grid to simplify boundary conditions. These “dead” ghost cells never stop and make it simple to compute edge and corner cells without the need for extra setup. By avoiding extra boundary checks for cells on the grid's border, this design minimizes conditional statements, which can impede calculation speed.

Additionally, every grid cell is iterated over by the program, which also verifies the condition of each of the eight adjacent cells. The status of the cell is updated in accordance with the game rules based on the total number of live neighbors. By pre-calculating indices and utilizing array offsets, the computation of neighbors is accomplished quickly. By doing this, complicated if-statements that might raise the time complexity are avoided.

Furthermore, to determine whether any cells change throughout a generation, a change flag is utilized. The simulation ends early, saving computing time in situations when the grid remains static or oscillatory, if no changes take place (i.e., the grid finds a stable configuration).

Testing Plan:

To ensure that the game rules were successfully implemented, preliminary tests were run on small grids (5x5 and 10x10) with a manually selected initial configuration. Initial testing and final testing used a Lenovo Flex 5-15IIL05 Laptop (IdeaPad) - Type 81X3**,** which has a 10th Generation Intel® Core™ i5-1035G1 Processor, 8GB of RAM, Windows 11 Home OS, and a clock speed of 1.19 GHz, and the compiler used was GCC 11.4.0. Conway's Game of Life provided known stable patterns, such as oscillators and still life (such as the “block” and “blinker” patterns), which served as the basis for the expected results of these experiments. The program was tried for larger grid sizes to assess performance after correctness was proven. With maximum generation counts of 1000 and 5000, the grid sizes that were utilized were 1000x1000, 5000x5000, and 10,000x10,000. Every test was run thrice, and the mean execution duration was noted. During these tests, print statements were commented out to prevent needless I/O latency. The entire program execution time was used to gauge performance. This was accomplished by timing the program from the beginning of the first generation to the end of the last generation using the C sys/time header file.

Regarding the test cases listed in Table 1, each test case yielded a different set of data, but as the grid size and maximum generation count grew, there was a discernible general increase in the average time required to finish the Game of Life simulation. With a maximum production limit of 1000 cells, the program took about 29.72 seconds to finish for the 1000 x 1000 cell universe. The comparatively short execution duration can be ascribed to the reduced grid size, which facilitates effective memory access and expedited cell updates. Further cutting down on computation time was the early termination condition, which came into play at generation 980 when the grid stabilized. The number of cell interactions grows quadratically with grid size, resulting in noticeably slower runtimes. For instance, the execution time skyrocketed to 832.16 seconds when the grid size was extended to 5000 x 5000, and the maximum generation count was set to 1000. This increase is expected since it takes longer to compute the neighboring states of each cell in a larger grid and to update the grid for every generation. With a maximum generation of 5000, the results for the 10000 x 10000 grid indicated a sharp rise in runtime to 18277.20 seconds. This is because the longer number of generations and the greater grid size increase computing complexity. Execution time increases significantly in big grids due to the sheer number of data involved, even with improvements like ghost cells and early termination.

Test Cases:

Test Case #	Problem Size	Max. Generations	Average Time Taken: HW 1 Results (seconds)	Average Time Taken of ASC Cluster: Intel Compiler (seconds)	ASC Cluster: Intel Compiler Options*
1	1000x1000	1000	29.7277	3.455576	8gb Memory Limit
2	1000x1000	5000	166.824	17.278363	8gb Memory Limit
3	5000x5000	1000	832.167	86.397196	8gb Memory Limit
4	5000x5000	5000	4519.59	1730.293398	8gb Memory Limit
5	10000x10000	1000	2085.45	1730293398	8gb Memory Limit
6	10000x10000	5000	18277.2	1729.33304	8gb Memory Limit

Table 1: HW1 vs Intel Compiler Performance Results

*Note: Intel Compiler used the ASC Cluster “class” queue, 1 core, and time limit of 12 hours.

Test Case #	Problem Size	Max. Generations	Average Time Taken of ASC Cluster: Intel Compiler (seconds)	Average Time Taken of ASC Cluster: GNU Compiler (seconds)	ASC Cluster: Intel Compiler Options*	ASC Cluster: GNU Compiler Options*
1	1000x1000	1000	3.455576	20.924215	8gb Memory Limit	8gb Memory Limit
2	1000x1000	5000	17.278363	104.621228	8gb Memory Limit	8gb Memory Limit
3	5000x5000	1000	86.397196	523.013584	8gb Memory Limit	10gb Memory Limit
4	5000x5000	5000	1730.293398	10473.271821	8gb Memory Limit	10gb Memory Limit
5	10000x10000	1000	1730293398	10468.494702	8gb Memory Limit	12gb Memory Limit
6	10000x10000	5000	1729.33304	10475.545007	8gb Memory Limit	12gb Memory Limit

Table 2. Intel Compiler vs GNU Compiler Results on ASC Cluster

*Note: Both the Intel and GNU Compiler used the ASC Cluster "class" queue, 1 core, and time limit of 12 hours.

Analysis and Conclusions:

The objectives of simulating and optimizing the Game of Life were effectively accomplished by this project. The running time of the program is O(N^2 * number of generations) because the "gameOfLife" function is called repeatedly by the main function up to a maximum number of generations. There are nested loops iterating over the N*N board inside the "gameOfLife" function. O(N^2 * number of generations) is the total time complexity as a result. The application operates effectively on grids up to 10,000 x 10,000 cells and accurately applies the game's rules. Particularly for higher grid sizes, the utilization of ghost cells and early termination optimization contributed to a reduction in computing complexity. The performance outcomes were as anticipated. Longer execution times were obtained for larger grids and higher generation counts; the execution time scaled roughly quadratically with grid size. For grids that reached stable states before reaching the maximum generating count, the early termination technique greatly enhanced performance. Large grid sizes could be handled by the software without the need for manual memory management because of the usage of an integer matrix of NxN size, which enabled dynamic memory allocation.

Regarding the results of Table 1, there is a significant difference between the times measured in Homework 1 and the Intel Compiler. For example, in test case 1, the HW 1 result is 29.7277 seconds, whereas the Intel Compiler took only 3.455576 seconds. For larger grids (e.g., 10,000 x 10,000 with 5,000 generations), the HW 1 result was 18,277.20 seconds, while the Intel Compiler completed it in 1,729.33304 seconds. The Intel Compiler is known for aggressive optimizations, especially with numerical code, memory access, and SIMD (Single Instruction, Multiple Data) instructions. It utilizes advanced optimization techniques (e.g., vectorization) that can significantly reduce execution times, which likely explains the drastic performance improvement. The Homework 1 tests were likely performed on a local machine (Lenovo Flex 5 laptop), while the Intel Compiler results were obtained from an ASC Cluster, benefiting from a more powerful and optimized hardware environment, such as using a high-performance server with ample resources (8GB memory limit). Regarding the results of Table 2, comparing the two compilers on the ASC Cluster, the Intel Compiler consistently outperformed the GNU Compiler across all test cases. For example, test case 1 (1000 x 1000, 1000 generations) took 3.455576 seconds with the Intel Compiler, while it took 20.924215 seconds with the GNU Compiler, and test case 6 (10,000 x 10,000, 5,000 generations) took 1,729.33304 seconds with the Intel Compiler but 10,475.545007 seconds with the GNU Compiler. Both compilers had similar memory limits (8GB-12GB), but the Intel Compiler likely had more sophisticated optimization flags enabled, such as aggressive loop unrolling and SIMD vectorization. The GNU Compiler may have been more conservative with optimizations, or the specific flags used were not as effective in maximizing performance on this workload. The GNU Compiler used higher memory limits (up to 12GB for the largest grids), but this did not translate into faster execution times. The Intel Compiler may have better memory management, reducing cache misses and improving memory access efficiency, which led to faster execution despite similar memory constraints. These differences highlight the importance of compiler choice and optimization strategies when working with high-performance computing tasks.

Future enhancements could include using vectorization, memory optimization, and parallelization to raise the program's overall quality. Primarily, using OpenMP for multithreading should enhance grid performance for larger grids, particularly for higher generation counts. Further memory optimization methods, like tiling or employing a sparse array, may also be able to lower memory overhead and improve cache efficiency, particularly for large grids with low living cell densities. Compiler optimizations may also be enabled to further accelerate performance by utilizing SIMD instructions.

In commemoration, I would pay extra attention to streamlining the grid's memory layout and lowering memory access overhead for larger grids. In order to make better use of contemporary multi-core computers, I would also investigate parallelization earlier in the design process.

In Peter J. Denning's "Exponential Laws of Computing Growth," he describes Moore's Law as a concept, first observed by Gordon Moore, which predicts that the number of components (such as transistors) on an integrated circuit will double approximately every two years, leading to exponential growth in computing power. Initially, it also implied increased speed due to smaller circuits allowing faster clock speeds. Additionally, Dennard Scaling, proposed by Robert Dennard in 1974, refers to the ability to scale down transistors while keeping power density constant. As transistors get smaller, the power consumption and heat dissipation should theoretically remain constant, allowing for increased speed without overheating the chip. However, this scaling reached its limit around 2000 due to heat dissipation issues, leading to the shift to multi-core processors. Moreover, the S-curve model describes the adoption of new technologies. Initially, adoption grows exponentially, but after reaching an inflection point, it slows down due to market saturation or technological limits. Businesses aim to jump to new technologies at the inflection point of the old ones to maintain growth. Furthermore, the S-curve model describes the adoption of new technologies. Initially, adoption grows exponentially, but after reaching an inflection point, it slows down due to market saturation or technological limits. Businesses aim to jump to new technologies at the inflection point of the old ones to maintain growth. Is Moore's Law dead? Moore's Law, in its original form, faces challenges due to physical limitations like heat dissipation, electron tunneling, and the economic costs of continued transistor shrinking. However, exponential growth in computing is likely to continue through **technology jumps**, where new technologies (e.g., 3D fabrication, carbon nanotubes, quantum computing) replace older ones. According to the authors, exponential growth occurs at three levels in the computing ecosystem. At the chip level, initially driven by Moore's Law, exponential growth at the chip level is supported by innovations such as multicore processors and power-saving techniques. At the system level, improvements in system architecture (e.g., parallelism, memory interconnects) allow for exponential growth in system performance. At the community level, the rapid adoption of new technology by user communities enables exponential growth in usage, which drives demand for more powerful chips and systems. The authors suggest that exponential growth is likely to continue for the next few decades through technology jumping—the shift to newer, more powerful technologies when the current ones reach their limits.

References:

1. Code sample matrixPassingValues.c, Md Kamal Chowdhury, September 12, 2024
2. Code sample life.c, Md Kamal Chowdhury, September 24, 2024
3. "Play John Conway's Game of Life." Playgameoflife.com, playgameoflife.com/. Accessed September 10, 2024

Github Link: https://github.com/gdprasad1201/CS-581-HPC-Fall-2024