Parallelizing Conway's Game of Life with OpenMP and CUDA on Talapas

Hritvik JV, Kaegan Kkoski

***Abstract*—Conway's Game of Life (GoL) is a two-dimensional cellular automaton whose simple local rules produce interesting emergent patterns. From a parallel computing perspective, GoL is also a nearly ideal example of a regular, data-parallel stencil computation: the next state of each cell depends only on a fixed neighborhood in the previous time step. This project uses GoL as a case study for parallel programming on the Talapas high-performance computing cluster at the University of Oregon. We implemented four versions in C: a serial baseline, a shared-memory parallel version using OpenMP, a GPU-accelerated version using CUDA and a distributed system using MPI. We then performed a strong scaling study of the OpenMP implementation and compared performance against the serial baseline and the CUDA implementation for a large grid. For a 4096×4096 grid evolved for 100 generations, the OpenMP implementation achieves approximately $27\times$ speedup using 32 threads compared to the single-threaded baseline, with about 85% parallel efficiency. The CUDA implementation further reduces the runtime to roughly 0.11 seconds, corresponding to about $4.3\times$ speedup over the 32-thread OpenMP run and about $116\times$ speedup over a single CPU thread for this configuration.**

I. Introduction

Conway's Game of Life (GoL) is a cellular automaton defined on a discrete two-dimensional grid of cells, each of which can be either "alive" or "dead" [1]. Time advances in discrete steps, and at each step the state of every cell is updated simultaneously based on the states of its eight neighbors. Despite the simple local rules, GoL is well-known for exhibiting complex emergent patterns such as gliders, birds, worms, oscillators, and chaotic behavior.

From a parallel computing perspective, GoL is an example of a stencil computation: each output cell is influenced by on a fixed pattern of cells in the input grid. This structure leads to a highly data-parallel workload. At each time step, the update of each cell is independent given the previous grid, which makes GoL a good fit for multi-core CPUs and GPU accelerators.

Goals:

1) Design and implement efficient serial, OpenMP, MPI and CUDA versions of GoL in C.
2) Evaluate strong scaling of the OpenMP version on a Talapas compute node (varying thread count at fixed problem size)
3) Compare performance of the best OpenMP configuration against a CUDA implementation on a GPU node.
4) Analyze communication overhead and perform weak scaling of MPI implementation

II. Methodology

This section describes the problem formulation, the serial baseline implementation, the OpenMP parallelization, the MPI implementation, the CUDA GPU implementation, and the experimental setup on Talapas.

A. Problem Definition

The simulation uses an $N \times N$ grid of cells. Let $g_t(y, x) \in \{0, 1\}$ denote the state of cell (y, x) at time step t, where 0 represents a dead cell and 1 represents a live cell. The neighborhood of (y, x) is its eight surrounding cells in the Moore neighborhood. The standard GoL rules are:

- **Survival:** If $g_t(y, x) = 1$ and the number of live neighbors is 2 or 3, then $g_{t+1}(y, x) = 1$.
- **Underpopulation & Overpopulation:** If $g_t(y, x) = 1$ and the number of live neighbors is less than 2 or greater than 3, then $g_{t+1}(y, x) = 0$.
- **Reproduction:** If $g_t(y, x) = 0$ and the number of live neighbors is exactly 3, then $g_{t+1}(y, x) = 1$.

The simulation uses toroidal boundary conditions: the grid wraps around at the edges so that neighbors of border cells are computed modulo N in each dimension. The simulation is run for G generations (time steps) starting from a random initial state where each cell is alive with probability $p \approx 0.3$, using a fixed random seed for reproducibility.

The main performance experiments in this project use:

- Grid size: $N = 4096$ (approximately 16.7 million cells).
- Generations: $G = 100$.

B. Serial C Implementation

The serial implementation serves as the reference baseline. The core data structure is a simple grid:

```
typedef struct { int N; int *data; } Grid;
```
(1)

The grid is stored as a one-dimensional array of length $N \times N$; the cell at (y, x) is stored at index $y \cdot N + x$. Two grid instances are allocated: `curr` (current generation) and `next` (next generation).

The `step_grid` function performs one update:

1) Loop over all rows $y = 0 \ldots N - 1$.
2) For each row, loop over all columns $x = 0 \ldots N - 1$.
3) For each cell (y, x), count the number of live neighbors by iterating over $dy, dx \in \{-1, 0, 1\}$ and skipping $(0, 0)$, using wrap-around to handle boundaries.
4) Apply the GoL rules to determine the new state and store it in `next`.

After completing the nested loops, the pointers `curr.data` and `next.data` are swapped to avoid copying the entire grid. The main function initializes the grid randomly, optionally prints the initial and final states for small N, and measures the execution time of the G calls to `step_grid` using `gettimeofday`. This runtime is reported as the serial baseline T_1.

C. OpenMP Parallel Implementation

The OpenMP implementation reuses the same `Grid` struct, random initialization, and update rules. The parallelization focuses on the `step_grid` function. The key observation is that updating each cell in the next grid depends only on reads from the current grid, and that each element of the next grid is written exactly once. Therefore, if `curr` is treated as read-only, the next grid can be computed in parallel without data races as long as each cell is assigned to exactly one thread.

In the OpenMP version, the outer loop over rows is annotated with:

```
#pragma omp parallel for schedule(static)
```
(2)

This directive instructs OpenMP to:

- Fork a team of `OMP_NUM_THREADS` threads.
- Divide the iterations of the outer loop (over y) among these threads using a static schedule, so each thread gets a contiguous block of rows.
- Run the inner loop over x serially within each thread for its assigned rows.
- Synchronize threads at the end of the loop before returning.

D. CUDA GPU Implementation

The CUDA implementation targets a single NVIDIA GPU on Talapas and The host code performs the following steps:

1) Allocate and initialize the host grid `h_grid` of length $N \times N$, with each cell alive with probability 0.3.
2) Allocate two device arrays `d_curr` and `d_next` using `cudaMalloc`.
3) Copy the initial grid from host to device with `cudaMemcpy`.
4) Configure a two-dimensional launch configuration using `dim3 blockDim(16,16);` `dim3 gridDim((N + blockDim.x - 1)/blockDim.x, (N + blockDim.y - 1)/blockDim.y);`
5) Create CUDA events to measure kernel execution time.
6) For each generation $t = 0 \ldots G - 1$, launch the kernel `gol_step_kernel<<<gridDim, blockDim>>>(d_curr, d_next, N);` synchronize, and swap `d_curr` and `d_next`.
7) Record the elapsed GPU time and wall-clock time, then copy the final grid back from device to host.

TABLE I
OPENMP STRONG SCALING ON TALAPAS FOR $N = 4096$, $G = 100$.

Threads P	Time T_P (s)	Speedup S_P	Efficiency E_P
1	13.245625	1.00	1.00
2	7.011376	1.89	0.94
4	3.415661	3.88	0.97
8	1.717138	7.71	0.96
16	0.882485	15.01	0.94
32	0.488025	27.14	0.85

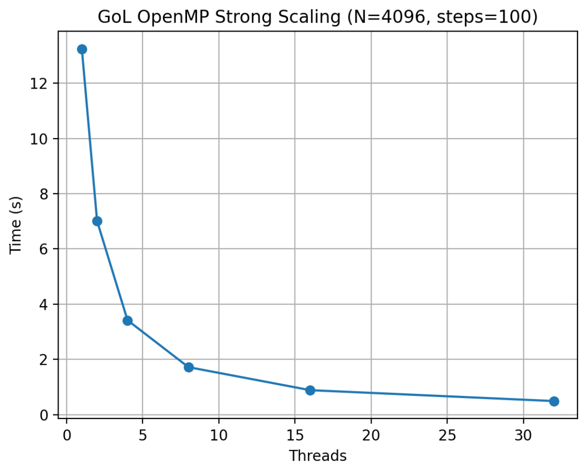

Fig. 1. OpenMP execution time vs. threads for $N = 4096$, $G = 100$.

Inside the kernel, each GPU thread computes its global coordinates (x, y) based on its block and thread indices:

$$x = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}, \quad (3)$$

$$y = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}. \quad (4)$$

Threads with $x \geq N$ or $y \geq N$ return immediately. For valid coordinates, the thread computes the linear index $i = y \cdot N + x$, counts neighbors in `curr` with toroidal wrap-around, applies the GoL rules, and writes the new state into `next[i]`. Conceptually, this assigns one cell to each GPU thread.

III. RESULTS

A. OpenMP Strong Scaling

For the OpenMP implementation, we performed a strong scaling study with $N = 4096$ and $G = 100$, varying the number of threads $P \in \{1, 2, 4, 8, 16, 32\}$. Let T_P denote the wall-clock time with P threads, $S_P = T_1/T_P$ the speedup, and $E_P = S_P/P$ the parallel efficiency.

Table I summarizes the measured runtimes and derived metrics.

Figure 1 shows the execution time as a function of thread count, and Figure 2 shows the corresponding speedup curve.

The results show nearly linear scaling up to 16 threads, with efficiencies above 90%. At 32 threads, the efficiency drops slightly to around 85%, which is still quite good. The non-linear behavior at higher thread counts can be attributed

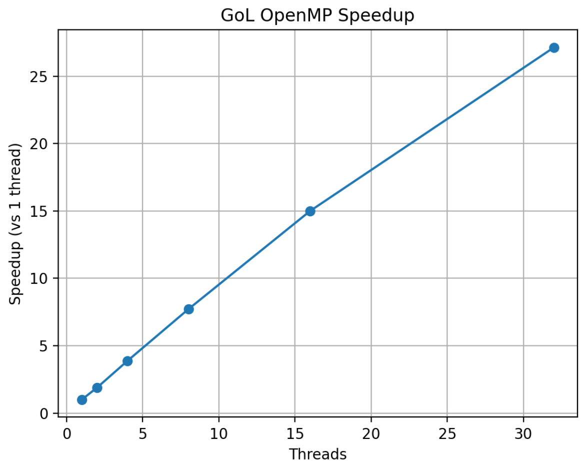

Fig. 2. OpenMP speedup vs. threads for $N = 4096$, $G = 100$.

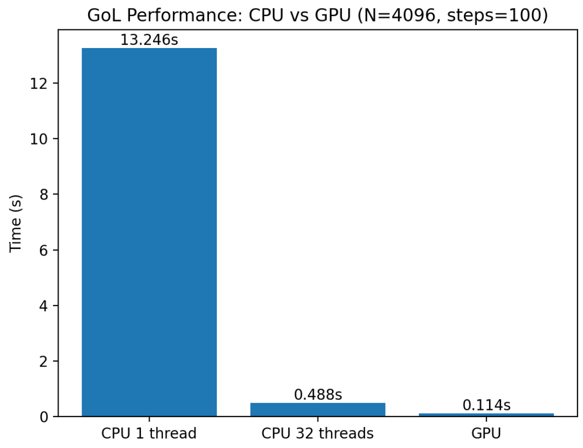

Fig. 3. CPU and GPU runtimes for $N = 4096$, $G = 100$.

TABLE II
CPU VS. GPU PERFORMANCE FOR $N = 4096$, $G = 100$.

Platform	Time (s)	Speedup vs 1 CPU	Speedup vs 32 CPU
CPU, 1 thread	13.2456	1.00	–
CPU, 32 threads	0.4880	27.14	1.00
GPU (CUDA)	0.1143	115.93	4.27

to overheads such as OpenMP thread management, memory bandwidth saturation, cache effects, and any serial portions of the code.

B. CPU vs. GPU Performance

To compare CPU and GPU performance, we fixed the problem size at $N = 4096$ and $G = 100$ and measured three configurations:

- Serial CPU baseline (1 thread).
- OpenMP CPU with 32 threads.
- CUDA GPU implementation on a single NVIDIA GPU.

The measured times are:

- CPU, 1 thread: $T_{1,\text{CPU}} = 13.245625$ s.
- CPU, 32 threads: $T_{32,\text{CPU}} = 0.488025$ s.
- GPU (CUDA): $T_{\text{GPU}} \approx 0.114257$ s (wall-clock).

The GPU implementation is approximately $116\times$ faster than the single-threaded CPU baseline and about $4.3\times$ faster than the already optimized 32-thread OpenMP implementation for this problem size. This is consistent with expectations: the GPU hardware provides many more execution units and higher memory bandwidth than a single CPU socket, and the regular, data-parallel structure of GoL allows the CUDA kernel to efficiently map each grid cell to a thread.

These speedups are observed for a relatively large grid where the cost of copying data to and from the GPU is amortized over many time steps. For smaller grids or very few time steps, the overhead of data transfers and kernel launches would likely reduce the advantage of the GPU.

C. Correctness Checks

To increase confidence in the correctness of the parallel implementations, small-grid tests were run (e.g., $N = 10$, $G = 5$) where the initial and final states were printed for:

- The serial implementation.
- The OpenMP implementation with one thread and with multiple threads.
- The CUDA implementation on a GPU node.

In all cases where comparisons were performed, the final states from the OpenMP and CUDA versions matched the serial output, indicating that the parallelization preserved the semantics of the GoL rules.

IV. Conclusion

This project used Conway's Game of Life as a case study for parallel computation, exploring both shared-memory and GPU-based parallelism. We implemented a serial C version, an OpenMP shared-memory version, and a CUDA version of the GoL simulation, all operating on a toroidal $N \times N$ grid.

The OpenMP implementation demonstrated strong scaling on a single compute node, achieving a speedup of approximately $27\times$ and parallel efficiency of about 85% when running with 32 threads on a 4096×4096 grid for 100 generations. This confirms that GoL, as a regular stencil computation, can effectively utilize multi-core CPUs with relatively simple OpenMP pragmas.

The CUDA implementation further accelerated the simulation by mapping each grid cell to a GPU thread and leveraging the massive parallelism of an NVIDIA GPU. For the same large grid and number of generations, the GPU implementation ran in about 0.11 seconds, corresponding to approximately $116\times$ speedup over the single-threaded CPU baseline and $4.3\times$ speedup over the 32-thread OpenMP configuration.

The project also highlighted several practical lessons:

- Even simple applications like GoL require careful attention to data layout, boundary conditions, and timing methodology to obtain meaningful performance measurements.

- OpenMP offers a low-effort path to shared-memory parallelism for regular loops and can deliver impressive speedups with minimal code changes.
- CUDA programming introduces additional complexity due to explicit device memory management and launch configuration, but the payoff can be substantial for highly parallel workloads.
- Profiling and comparing multiple implementations on the same hardware helps build intuition about where different programming models and architectures provide the most benefit.

The original proposal also envisioned weak scaling experiments and a communication-overhead analysis using MPI. Due to time constraints and the focus on getting robust OpenMP and CUDA implementations running on Talapas, the MPI implementation and those specific analyses were not completed. In future work, we would like to implement an MPI version of GoL that partitions the grid across processes and performs halo exchanges at each time step, revisit the weak scaling and communication-overhead analyses in the original proposal, and experiment with more advanced CUDA optimizations and extended models such as forest-fire cellular automata.

References

[1] M. Gardner, "Mathematical games: The fantastic combinations of John Conway's new solitaire game 'life'," *Scientific American*, vol. 223, pp. 120–123, 1970.

[2] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 4.5," Nov. 2015.

[3] NVIDIA Corporation, "CUDA C Programming Guide," Version 12.x.